Lambda Expressions in Java

Reference

Angelika Langer & Klaus Kreft

Angelika Langer Training/Consulting - www.AngelikaLanger.com

Lambda Expressions in Java - Reference

by Angelika Langer & Klaus Kreft - www.AngelikaLanger.com

Cover design by Angelika Langer

ISBN n.n

Pre-Release, November 16, 2013

Table of Contents

Preface

This is part II of a series of books on "Lambdas & Streams in Java 8". The series aims to provide comprehensive information about new language features in Java 8 (collectively referred to as "Lambdas") and new JDK abstractions for bulk operations on collections (collectively referred to as "Streams"). The series has four parts:

- Part I: Lambda Expressions in Java - Tutorial
- Part II: Lambdas Expressions in Java - Reference
- Part III: Streams - Tutorial
- Part IV: Streams - Reference

This is part II entitled "Lambdas Expressions in Java - Reference". It provides details regarding all aspects of lambda expressions, method and constructor references, functional interfaces, default interface methods, and static interface methods.

Prerequisites

Before you read this book you should be familiar with the basics of lambda expressions and the related new language features. This kind of basic understanding can be acquired by glossing over a tutorial such as part I of the series. Part I (The Lambda Tutorial) gives an overview of the new language features, briefly explains what they look like and what they can be used for. Naturally, the tutorial omits many details - which in turn this book (Part II - The Lambda Reference) aims to provide.

In addition you should be familiar with the basics of the stream API because many of the examples in this book use the stream API without explaining it in detail. A basic understanding of the stream API can be gained by reading a tutorial such as Part III (The Stream Tutorial) of the series. The JavaDoc of JDK 8 plus the Stream Tutorial provide enough information to easily comprehend the examples.

How to Use This Book

While the tutorial is best read cover to cover, the reference is suitable for selective reading. It can be consulted whenever you are interested in details regarding a certain aspect of lambdas.

In order to promptly find the information related to the topic at hand there you can use:

- the table of contents,
- the questions & answers section, and
- the index.

They point to the respective relevant sections that might contain the information you are looking for.

While the tutorial (Part I - The Lambda Tutorial) provides an introduction and overview, this book (Part II - The Lambda Reference) intends to provide details regarding lambdas and related language features.

For illustration of the level of detail that this book (Part II - The Lambda Reference) covers, let us consider the issue of type inference. It is a process that the compiler applies for lambda expressions and method references in order to determine their static type. Type inference is a detail that you usually need not care about because the compiler does it automatically behind the curtain. Most of the time you will not even notice that type inference happens at all. But, occasionally, you might run into a situation where automatic type inference fails and you receive error messages during compilation. Then you can use the lambda reference to learn about the principles of type inference so that you can figure out how to cope with a type inference failure.

Occasionally, the reference covers general information about certain language features before it gets to lambda expressions. This is in order to put the issues related to lambdas into perspective. An example is the section on "The Meaning of Names - Scopes, Shadowing, and Binding". It first explains the scoping rules in classic Java before it addresses the meaning of names in lambda expressions. Naturally, if you are already familiar with scopes and name binding in Java you might want to skip what you are familiar with and move forward to the lambda related information. As an aid a hint that points to the lambda related issues is provided at the beginning of the section.

Other sections discuss features might be useful in rare situation, but that do not exist in Java (because the language designers decides so). An alternative or work-around is provided, if available. Still, not everybody is interested reading about corner issues and absent features; for this reason there is a hint at the beginning of the respective sections that cleary states that "this is an esoteric corner issue". An example is the section on "Generic Lambda Expressions". They do not exist and you can method references or anonymous inner classes instead.

How This Book Is Organized

Chapter 1 is devoted to LAMBDA EXPRESSIONS.

It covers syntax details, explains lexical scoping and illustrates the use of names in lambda expressions, and discusses the use of `break`, `continue`, `return`, and `throw` statements used in lambda expressions. The section also points out that certain features are not supported in Java; this concerns recursive lambda expressions and generic lambda expressions.

Chapter 2 covers METHOD AND CONSTRUCTOR REFERENCES.

It illustrates the syntax variants for references to constructors, non-static and static methods and provides many examples thereof.

Chapter 3 covers FUNCTIONAL INTERFACES. It explains what a functional interface is and what distinguishes it from a regular interface. The role of functional interfaces in conjunction with lambda expressions and method/constructor reference is discusses. Special cases such as generic functional interfaces and intersections of function interfaces are mentioned.

Chapter 4 explains TARGET TYPING in general and type inference for lambda expressions and method/constructor references in particular. We take a look under the hood of the compiler and learn about poly expressions, of which lambda expressions are a special case. Target typing requires an inference context and for this reason lambda expressions and method/constructor references may only appear in a legal inference context. The chapter touches on all forms of inference contexts that are legal in Java and also discusses type inference failure.

Chapter 5 discusses the RUNTIME REPRESENTATION. This goes into the internals of representing lambda expressions in the JVM. It explains technical details such a s the lambda factory the use of the invokedynamic byte code instruction, the generation of bridge methods, and the serialization of lambda expressions.

Chapter 6 covers NON-ABSTRACT INTERFACE METHODS. Interfaces can have two types of non-abstract interface methods, namely default methods and static interface methods. Both are explained in this chapter. Via default methods, the debatable feature of multiple inheritance of implementation has sneaked into Java. The chapter discusses whether multiple inheritance with default methods is harmful or not.

Chapter 7 is devoted to PROGRAMMING WITH LAMBDAS. It discusses the execute-around-method pattern as an exampel of a

programming idiom that is convenient and elegant to use by beans of lambda expressions.

How to Contact the Authors

Additional information regarding the book series can be found at:

http://www.AngelikaLanger.com/Lambdas/Lambdas.html

You can address comments and questions about the book series to the authors using the contact form at:

http://www.AngelikaLanger.com/Forms/Lambda.html.

Acknowledgements

Many thanks to all colleagues, readers, and reviewers who took the time to read the material and provided constructive feedback. A number of individuals at Oracle patiently answered questions we posed regarding lambda expressions and streams. Our thanks to Brian Goetz, Maurizio Cimadamore, and Paul Sandoz.

Questions & Answers

Lambda Expressions

Syntax

The Meaning of Names - Scopes, Shadowing and Binding

The Meaning of Jumps and Exits

Recursive and Generic Lambda Expressions

Method and Constructor References

Functional Interfaces

What are functional interfaces needed for?
Functional interfaces are needed as the target type of a lambda expression or method / constructor reference.

functional interface 68

What is a functional interface?
An interface with one abstract method.

definition of functional interface 68

Can a functional interface have non-abstract methods?
Yes, it can have default methods and methods inherited from Object.

non-abstract methods in functional interfaces 69

What is the purpose of the @Functional Interface annotation?
It indicates that an interface is intended as a functional interface and triggers certain compiler checks.

@Functionalnterface annotation 71

Can functional interfaces be generic?
Yes.

generic functional interface 71

Are parameterizations of a generic functional interface still functional interfaces?
Yes. If we take a functional interface and replace its type parameters by concrete types the resulting parameterization is still a functional interface.

parameterization of generic interface 72

Is the raw type of a generic functional interface still functional?
Yes. If we drop the type parameters of a functional interface then the resulting raw type is a functional interface as well.

raw type of generic interface 72

Is the intersection of several functional interfaces functional, too?
Yes, if the intersection contains a single abstract method.

intersection of functional interfaces 74

Target Typing

What does the term "target type" mean? When an expression appears in a context, its type must be compatible with a type expected in that context. The expected type is called the *target type*.	**target type**	76
What is a standalone expression? An expression whose type is determined by the expression's content.	**standalone expression**	77
What is a poly expression and in which context may it appear? An expression whose type is context dependent.	**poly expressions**	78
What is a poly context? A context that provides information for inference of a poly expression's type.	**poly context**	79
How is the target type inferred if the "diamond operator" is used? By means of the left-hand side type of an assignment or a method's declared argument type.	**target typing for diamond operator**	80
How is the target type inferred when generic methods are invoked? By means of the left-hand side type of an assignment or a method's declared argument type.	**target types with generic method**	114
How is the target type inferred for a conditional operator expression? By means of the left-hand side type of an assignment or a method's declared argument type.	**target typing for conditional operator**	80
How is the target type of a method reference inferred? By means of the left-hand side type of an assignment or a method's declared	**target typing for method/constructor references**	84

argument type, or the target type of a cast, and comparison of the function descriptors.		
Do throws clauses matter in the target typing process? Yes, they must be compatible.	**target typing and checked exceptions**	86
Do return types matter in the target typing process? Yes, they must be compatible.	**target typing and return type**	89
How is the target type inferred for lambda expressions? By means of the left-hand side type of an assignment or a method's declared argument type, or the target type of a cast, and comparison of the function descriptors.	**target typing for lambda expressions**	89
What is an intersection type? A synthetic type that is a subtype of several supertypes.	**intersection type**	92
Can overloaded methods serve as method invocation context for a poly expression? Yes, they can, but they may lead to error messages due to ambiguity.	**target typing & overloading**	96
What can I do if I pass a lambda to an overloaded method and the compiler rejects it due to an ambiguity? Specify the lambda's arguments types explicitly.	**use explicit lambdas**	98
Is overloading generally discouraged in Java 8? No, it causes problems only in conjunction with lambdas, i.e. do not overload on different functional types	**avoid overloading on functional types**	105
Can generic functional interfaces be implemented by lambda expressions and method/constructor references?	**target typing & generic target types**	111

Yes.

What happens if the target type is a wildcard parameterization of a generic functional interface?
The compiler replaces the wildcard by its bound (or object if unbounded).

target typing & wildcards 111

Can a functional interface with a generic method be implemented by lambda expressions and method/ constructor references?
No, for lambda expressions. Yes, for method/constructor references.

target types with generic method 114

Non-Abstract Interface Methods

What is a non-abstract method in an interface?
A non-abstract interface method has an implementation. It is either a default method or a static method.

non-abstract interface method 117

Default Interface Methods

What are default methods intended for?
They permit adding functionality to existing interfaces without breaking the subclasses.

default methods 117

Can default methods be `private` or `protected`?
No, they are always `public`.

accessibility of default methods 119

Why can't default methods be `final`?
Because it could break existing subclasses.

final default methods 120

Do default methods add multiple inheritance to Java?
Yes, default methods in interfaces permit multiple inheritance of

multiple inheritance 120

functionality.

What is the "deadly diamond of death"?
An error-prone form of multiple inheritance.

deadly diamond of death 121

Is multiple inheritance in Java dangerous?
No.

peril of multiple inheritance 121

What are default methods used for?
Evolution of existing interfaces plus API development in general

programming with default methods 122

Can multiple inheritance involving default methods lead to conflicts and ambiguities?
Yes.

ambiguous default methods 127

Static Interface Methods

What is a static interface method?
A method in an interface with the static modifier and an implementation.

static inferface methods 135

How do static and default interface methods differ?
Like static class methods differ from non-static class methods: static method cannot access non-static members and static methods cannot be overridden.

difference static / default method 135

Can static methods be `private` or `protected`?
No, they are always `public`.

accessibility of static interface methods 138

How do static interface and static class methods differ?
Static interface methods are not inherited, must be invoked via their declaring interface type and must not be invoked via an instance.

difference static interface / class methods 138

Programming with Lambdas

How do I correctly use generic functional interfaces? When generic functional interfaces appear as arguments of operations they are often parameterized with a wildcard.	**wildcards in functional API design**	159

Lambda Expressions

Syntax

A lambda expression describes an anonymous function. Like a function it has a parameter list and a body. Let us start with a simplified version of the syntax. We will get to the detailed syntax later on.

```
LambdaExpression:
  LambdaParameters '->' LambdaBody

LambdaParameters:
  Identifier
  '(' ParameterList ')'

LambdaBody:
  Expression
  Block
```

A lambda expression consists of a parameter list, the arrow symbol `"->"`, and a body.

Here are a couple of examples of lambda expressions and their equivalent as a method with an arbitrary name (namely `nn` in the table below).

lambda expression	equivalent method
`() -> { System.gc(); }`	`void nn() { System.gc(); }`
`(int x) -> { return x+1; }`	`int  nn(int x) return x+1; }`
`(int x, int y)` `-> { return x+y; }`	`int  nn(int x, int y)` `{ return x+y; }`
`(String... args)` `->{return args.length;}`	`int  nn(String... args)` `{ return args.length; }`

`(String[] args)`
`-> {`
`  if (args != null)`
`  return args.length;`
`  else`
`  return 0;`
`}`
	`int  nn(String[] args)`
	`{`
	`  if (args != null)`
	`  return args.length;`
	`  else`
	`  return 0;`
	`}`

```
(Future<Long> f)
 -> { return f.get(); }
```

```
Long nn(Future<Long> f)
 throws ExecutionException,
      InterruptedException
{ return f.get(); }
```

These lambdas look similar to methods. The difference is that they do not have a name and neither the return type nor the throws clause is declared. Both are automatically inferred by the compiler.

Body

The lambda body can be a block of one or more statements. It can also be as simple as a plain expression.

Here are a couple of examples of lambda expressions with a body that consist of no more than an expression:

lambda expression	equivalent method
`() -> System.gc()`	`void nn() { System.gc(); }`
`(int x) -> x+1`	`int nn(int x) { return x+1; }`
`(int x, int y) -> x+y`	`int nn(int x, int y)` `{ return x+y; }`
`(String... args)` `-> args.length`	`int nn(String... args)` `{ return args.length; }`

`(String[] args)`
`->`
`(args!=null)? args.length : 0`
	`int nn(String[] args)`
	`{`
	`  return (args != null)`
	`  ? args.length: 0;`
	`}`

`(Future<Long> f)`
`-> f.get()`
	`Long nn(Future<Long> f)`
	`throws ExecutionException,`
	`InterruptedException`
	`{ return f.get(); }`

Here are a couple of counter examples, namely lambda expressions that do no compile:

lambda expression	at compile-time
`(int x) -> return x+1`	error: not a statement The return keyword can only appear in a statement and statements must be terminated by a semicolon.
`(int x) -> return x+1;`	error: illegal start of expression A statement can only appear in a block, i.e. the statement must be enclosed in curly braces.
`(int x)  -> { return x+1; }`	fine
`(Future<Long> f) -> { f.get() }`	error: ';' expected In a block only statements are permitted; a plain expression is illegal.
`(Future<Long> f)` `-> { f.get(); }`	error: missing return value In a block there must be a return statement if a return value is exptected.
`(Future<Long> f)` `-> { return f.get(); }`	fine

Parameter List

There are a couple of variations of the parameter list. The parameter types of a lambda expression may either all be declared or all inferred. The inferred types are derived from the context in which the lambda expression appears.

Single Parameter with Inferred Type

If the parameter list consists of exactly one parameter then it can be reduced to an identifier, i.e. the name of the parameter.

Examples of a parameter list reduced to a single identifier:

lambda expression	at compile-time
`x -> x+1`	fine
`args -> args.length`	fine
`f -> { return f.get(); }`	fine
`f -> f.get()`	fine
`int x -> x+1`	error: illegal If the parameter type is specified then the enclosing parenthesis are mandatory.
`(int x) -> x+1`	fine
`x,y -> x+y`	error: illegal If there is more than one parameter then the enclosing parenthesis are mandatory.
`(int x,int y) -> x+y`	`fine`

The notation without a declared type requires that the compiler can infer the parameter type from the context in which the lambda expression appears.

The syntax permits further variations, among then an empty parameter list, a parameter list without declared parameter types, and a variable parameter list. Here is the parameter list's detailed syntax:

```
LambdaParameters:
  Identifier
  '('              InferredFormalParameterList             ')'
  '(' FormalParameterListopt ')'
InferredFormalParameterList:
  Identifier
  InferredFormalParameterList ',' Identifier
```

```
FormalParameterList:
  LastFormalParameter
  FormalParameters ',' LastFormalParameter

FormalParameters:
  FormalParameter
  FormalParameters, FormalParameter

FormalParameter:
  VariableModifiersopt Type VariableDeclaratorId

LastFormalParameter:
  VariableModifiersopt    Type    '...'    VariableDeclaratorId
  FormalParameter
```

Multiple Parameters with Inferred Types

The `InferredFormalParameterList` is a list of parameters without type specifications. The enclosing parentheses are mandatory. The declared type can only be omitted when the compiler can infer the parameter type from the context in which the lambda expression appears.

Examples of the `InferredFormalParameterList`:

lambda expression	**at compile-time**
`(int x)  -> x+1`	fine
`int x  -> x+1`	error: illegal If the parameter type is specified then the enclosing parenthesis are mandatory
`(x)  -> x+1`	fine
`x  -> x+1`	`fine` (because there is only one parameter)
`(int x,int y) -> x+y`	fine
`int x,int y  -> x+y`	error: illegal If the parameter type is specified then the enclosing parenthesis are mandatory

`(x,y) -> x+y`	fine
`x,y -> x+y`	error: illegal If there is more than one parameter then the enclosing parenthesis are mandatory.

Regular Parameters with Declared Types

The regular `FormalParameterList` is a list of parameters as we know it from methods. The enclosing parentheses are mandatory. The parameter list can be empty, in which case it consist of empty brackets. It can contain one or more parameters with a declared type. If one parameter has a declared type, then all parameters must have a declared type. Mixing inferred and declared types is illegal. The declared type can have modifiers like `final` for instance. The last parameter may be a variable parameter, also know as "varargs" parameter like `String...` for instance.

Examples of the regular `FormalParameterList`:

lambda expression	at compile-time
`() -> 42`	fine
`(int x) -> x+1`	fine
`(int x,int y) -> x+y`	fine
`(String fmt, Object... args) -> String.format(fmt,args)`	fine
`(int x, y) -> x+y`	error: illegal Mixing inferred and explicit parameter types is illegal.
`(final int x) -> x+1`	fine
`(final x) -> x+1`	error: illegal Only explicit parameter types can have

modifiers.

Return Type and Throws Clause

Different from a method a lambda expression does neither declare a return type nor a throws clause. Both the lambda's return type and its throws clause are always automatically inferred by the compiler from the context in which the lambda expression appears.

The Meaning of Names - Scopes, Shadowing, and Binding

Lambda expressions are frequently compared to methods on the one hand and anonymous inner classes on the other hand. The syntax of lambda expressions is similar to the syntax of methods, as we have seen in the section on "Syntax". At the same time, lambda expressions are used in places where traditionally anonymous inner classes were used (see the section on "What are lambda expressions?" in the *Lambda Tutorial* document for examples).

The key difference between the "new" lambda expressions and "old" methods and anonymous inner classes is that classes and methods have their own scopes. Methods and classes can declare names that are in the respective method or class and shadow corresponding names in enclosing scopes. Lambda expressions, in contrast, are part of their enclosing scope (i.e., they are NOT scopes of their own). Names declared in a lambda expression contribute to the enclosing scope; they never shadow names in the enclosing scope.

In the following we will discuss what a variable name means in the body of a lambda expression, in a method body, or in an anonymous inner class, and what happens if the same name is declared in different, potentially nested scopes.

Note: In the subsequent sections we first explain scopes, name binding, and shadowing in general before we look into name binding issues related to lambda expressions. If you are already familiar with the concept of scopes, name binding, and shadowing, feel free to skip the subsequent sections and continue with the section on "Lexical Scoping for Lambda Expressions".

Scopes

Java has a notion of *scopes*. A scope is the part of the program text within which a name can be referred to without any qualification, i.e., by using the simple name. Examples of scopes are classes and methods.

For instance, if we define a static field in a class then the scope of this field is the entire class; we can use the field's name everywhere within the

class without needing any qualification. If we refer to the name from outside of the class, in contrast, we need to qualify the name.

Example of a name defined in class scope:

```
public class Container {
  public static final int MAX_CAPACITY = 1024;
  public Container() {
    ... = new Object[MAX_CAPACITY];        // simple name
  }
}
public class Test {
  public static void main(String... args) {
    if (size < Container.MAX_CAPACITY)     // qualified name
      ...
  }
}
```

The scope of the name `MAX_CAPACITY` is the declaring class `Container`. The name `MAX_CAPACITY` can be used inside the class scope without qualification and must be qualified (by the name of its declaring class `Container`) outside of the class scope, e.g. when used in class `Test`.

Scopes are distinct, i.e., the same name can appear in different scopes and refer to different entities. For instance, another class `Sequence` might also declare a field named `MAX_CAPACITY`. We would then have two separate fields, namely `Container.MAX_CAPACITY` and `Sequence.MAX_CAPACITY`.

```
public class Container {               // declaring scope #1
  public static final int MAX_CAPACITY = 1024;
  ...}
}
public class Sequence {                // declaring scope #2
  public static AtomicInteger MAX_CAPACITY
    = new AtomicInteger();
  ...
  }
}

public class Test() {                  // using scope
  public static void main(String... args) {
    if (Container.MAX_CAPACITY
                <= Sequence.MAX_CAPACITY.get())
      ...
  }
}
```

Nested Scopes

Matters are simple as long as the scopes are clearly separated like in the example above. But, what happens if scopes are nested, e.g., if one class appears in the scope of another class, and both classes declare the same name? Depending on the situation, two effects can occur:

- *shadowing*: the inner name shadows the outer name, or
- *binding*: the inner name is bound to the outer name.

Shadowing means that use of the simple name in the inner scope refers to the inner entity and the outer entity is hidden, i.e., shadowed. In order to refer to the shadowed outer entity a qualification is needed.

Binding means that use of the simple name in the inner scope refers to the outer entity and there is no separate inner entity. Both the inner and the outer name refer to the same entity declared in the outer scope.

In the following we will look into different situations in which names are shadowed. We will first consider scopes that are nested into a class and then scopes that are nested into a method.

Names in a Class Scope

In this section we discuss name binding issues in scopes that are nested into a class.

A class is a scope. It may contain other entities, such as methods, nested or inner classes, or lambda expressions. Some of these entities are also scopes, i.e., we have nested scopes in these situations. The outer and the inner scope might define the same name. How does name binding work in case of scopes nested into a class scope?

We will discuss the following situations:

- a class defined in another class
- a method defined in a class
- a lambda expression defined in a class

Nested Scopes#1: A Class Nested into another Class

What happens if classes are nested, e.g., if one class appears in the scope of another class, and both classes declare the same name? Then the inner name shadows the outer name.

Example of a class nested into a class scope:

```
public class Container {                                          //
outer scope
  public static final int MAX_CAPACITY = 1024;
  public Container() {
   ... = new Object[MAX_CAPACITY];
  }
  private static class NestedSequence {                           //
inner scope
    public static AtomicInteger MAX_CAPACITY;
    public NestedSequence() {
      MAX_CAPACITY
       = new AtomicInteger(Container.MAX_CAPACITY);
    }
  }
}

public class Test() {                                             //
unrelated using scope
  public static void main(String... args) {
    if (Container. MAX_CAPACITY
        <= Container.NestedSequence.MAX_CAPACITY.get())
       ...
  }
}
```

Class `NestedSequence` is declared in the scope of the enclosing class `Container`. Both classes declare a field `MAX_CAPACITY`. When the simple name `MAX_CAPACITY` is declared in the nested class it shadows the name `MAX_CAPACITY` from the enclosing class. If we want to refer to the `MAX_CAPACITY` field from the enclosing scope we need to use the qualified name `Container.MAX_CAPACITY`.

Nested Scopes#2: A Method Nested into a Class

Methods, too, are scopes. If we define a name in a method then the scope of this name is the method including the method's parameter list. Methods always have an enclosing scope, namely the scope of their declaring class. In analogy to nested classes, names in a method shadow names declared in the enclosing class.

Example of methods nested into a class scope:

```
class Sequence {                          // outer scope
  private int capacity = 0;

  public Sequence(int capacity) {         // inner scope #1
    int tmp = capacity;
    ...
```

```
    this.capacity = tmp;
  }
  public void resize(int capacity) {       // inner scope #2
    int tmp = capacity;
    ...
    this.capacity = tmp;
  }
}
```

Both the constructor and the `resize` method declare the name `tmp`. Each method is a scope of its own and the two entities named `tmp` do not collide.

The enclosing class declares the name `capacity` and both the constructor and the `resize` method declare the same name `capacity`. We have three scopes (class, constructor, and method) and therefore we have three entities named `capacity`. The use of the simple name `capacity` inside the constructor or method shadows the name `capacity` declared in the enclosing class. If we want to refer to the enclosing `capacity` inside a constructor of method we must use the qualified name `this.capacity`.

Nested Scopes#3: A Lambda Expression Nested into a Class

When lambda expressions appear on the class level, the rules for the names they declare are the same as for methods: names in the lambda expression shadow names declared in the enclosing class.

Example of lambda expression nested into a class scope:

```
class Sequence {                           // outer scope
  private int capacity = 0;

  public void resize(int capacity) {       // inner scope #1
    int tmp = capacity;
    ...
    this.capacity = tmp;
  }
  private IntConsumer resizer = (int capacity) -> {
                                           // inner scope #2
    int tmp = capacity;
    ...
    this.capacity = tmp;
  };
}
```

The lambda expression declares local entities named `tmp` and `capacity` of its own and uses the qualified name `this.capacity` if it needs a reference to the enclosing class's entity.

The principle is always the same. If we have an outer scope (a class) and an inner scope (a nested type, method, or lambda expression) and the outer and the inner scope declare the same name, then both names denote separate entities. The inner name shadows the outer name and we can distinguish between the entities by using qualified names. The qualifier is either a type name (e.g. `Container.MAX_CAPACITY`) or an expression (e.g. `this.capacity`).

Note, that a name declared in a class is visible in the entire class. Its declaration can be placed before or after its use, i.e., the declaration of the name need not precede its use.

Example of using a name before declaring the name:

```
class Sequence {
  public void resize(int capacity) {
    int tmp = capacity;
    ...
    this.capacity = tmp;                    // usage
  }
  private IntConsumer resizer = (int capacity) -> {
    int tmp = capacity;
    ...
    this.capacity = tmp;                    // usage
  };

  private int capacity = 0;                 // declaration
}
```

Names in a Method Scope

In this section we discuss name binding issues in scopes that are nested into a method.

Since methods cannot be defined inside other methods, the only scopes that can be nested into a method are local or anonymous inner classes and lambda expressions.

We will look into the following situations:

- a class defined in a method
- a lambda expression defined in a method

Nested Scopes#4: A Class Nested into a Method

First, we consider classes that appear in the scope of a method. If both the enclosing method and the local or anonymous inner class declare the same name, then we have two entities with the same name in different nested scopes. Like top level and nested classes, local and anonymous classes are scopes.

Example of a local class nested into a method scope:

```
public void reverse_sort(Comparator arg) {     // outer scope
  Comparator cmp = null;
  class ReverseComparator implements Comparator {
                                                 // inner scope
    private Comparator cmp;
    public ReverseComparator() {
      cmp = arg;
    }
    public ReverseComparator(Comparator arg) {
      cmp = arg;
    }
    public int compare(Object lhs, Object rhs)  {
      return cmp.compare(rhs,lhs);
    }
  }
  cmp = new ReverseComparator() ;
  ...
}
```

In the code snippet above the `reverse_sort` method declares a local variable named `cmp` and the local class `ReverseComparator` defines a field named `cmp`. Since the local class is a scope, we end up with two entities in two different scopes that have the same name. Using its simple name in the inner scope shadows the name in the outer scope. This time there is no qualification that would permit reference to the outer entity from the inner scope. There is simply no qualifier for a local variable such as the `cmp` variable in method `reverse_sort`.

The example above also illustrates name binding. The outer scope, i.e. the `reverse_sort` method, declares a variable named `arg`. The no-argument constructor of class `ReverseComparator` does not declare a variable named `arg` of its own, but uses the name `arg` to refer the outer scope variable `arg`. Before Java 8, this kind of name binding was only permitted if the outer scope variable was declared as `final`. Since Java 8, the name binding is allowed without the `final` declaration. The bound variable, however, must be *effectively final*, which means that you need not explicitly declare it as `final`, but the compiler implicitly treats it as `final` and issues error messages for every attempted modification.

The other constructor of class `ReverseComparator` does declare a variable named `arg` of its own. In this constructor the simple name `arg` refers to the local variable `arg` and not to the outer scope variable `arg`. Here we have shadowing instead of binding. (In practice, it is advisable to use different names in order to avoid any confusion regarding the meaning of each name. We are using colliding names in the example solely for the sake of illustration.)

In the example above we have been using a local class as the inner scope. The same shadowing and binding occurs for anonymous inner classes. Here is the same example with an anonymous instead of a local class.

Example of an anonymous class nested into a method scope:

```
public void reverse_sort(Comparator arg) {    // outer scope
  Comparator cmp = null;
  cmp = new Comparator() {                    // inner scope
    private Comparator cmp;
    { // initializer
        cmp = arg;
    }
    public int compare(Object lhs, Object rhs)  {
        return cmp.compare(rhs,lhs);
    }
  };
  ...
}
```

The name `cmp` declared in the anonymous inner class shadows the same name declared in the enclosing method. The name `arg` is not declared in the anonymous inner class and refers to the outer entity, which is implicitly `final`. Note, in this version of the example the inner variable named `cmp` can be eliminated; is it used solely for illustration.

Nested Scopes#5: A Lambda Expression Nested into a Method

If we use lambda expressions inside a method we find that they are not scopes of their own, different from local and anonymous inner classes. Instead, a lambda expression is part of the scope in which it appears. This is called *lexical scoping* and is discussed in the next section.

Lexical Scoping for Lambda Expressions

If a lambda expression is defined in a method and declares names that already exist in the enclosing method, then the compiler issues an error message. This is because a lambda expression is not a scope, but is part of its enclosing scope. (This is called *lexical scoping*.)

Example of a lambda expression nested into a method scope:

```
public void reverse_sort(Comparator arg) {    // outer scope
  Comparator cmp = null;
  cmp = (Object lhs, Object rhs)  -> {     // NO inner scope
    Comparator cmp;       // error: name cmp already defined
    cmp = arg;
    return cmp.compare(rhs,lhs);
  };  ...
}
```

The example illustrates that the lambda expression is part of the method scope in which it appears. When the name `cmp` is declared in the lambda expression the compiler considers it an attempt to define a name that already exists in the current scope and issues an according error message.

This is a fundamental difference compared to local and anonymous inner classes. While classes are scopes of their own, lambda expressions are part of their enclosing scope and do not define names of their own. **Every name used inside the lambda expression has the same meaning as in the enclosing scope.** This is one reason why lambdas in Java are called lambda expressions: they are expressions (in contrast to classes or methods). Regarding scopes and the meaning of names lambda expressions behave like expressions: they are part of the enclosing scope and contribute to it.

The name binding works the same way as it does for local and anonymous inner classes. If a name declared in the enclosing scope is used inside the lambda expression then it refers to the outer entity. In the example, use of the name `arg` in the lambda refers to the `arg` variable in the enclosing method scope.

Let us fix the error in the lambda expression above. We can use a name different from `cmp` and thereby eliminate the name collision. Since the lambda expression in our example does not even need the `cmp` variable we can drop it altogether.

Example identical to the one before, but without an error:

```
public void reverse_sort(Comparator arg) { // outer scope
  Comparator cmp = null;
  cmp = (Object lhs, Object rhs)  -> {     // NO inner scope
    return arg.compare(rhs,lhs);
  };
  ...
}
```

Note, that a name declared in a method must be declared *before* its use. This is different from the declaration of names in class scope. A name defined in a class scope is visible in the entire class; it can be declared after is has been used. This is not permitted for names in methods. A name declared in a method is visible from its declaration until the end of the scope.

This means that a name defined in a lambda expression only collides with identical names of the enclosing method if the colliding name has been declared before it appears in the lambda. There is no collision if the name is declared after the lambda.

Example that demonstrates the above:

```
public void reverse_sort(Comparator arg) {
  Comparator cmp = null;
  cmp = (Object lhs, Object rhs)  -> {
    Comparator tmp = arg;          // fine; no name collision
    return tmp.compare(rhs,lhs);
  };
  Comparator tmp = arg;
  cmp = (Object lhs, Object rhs)  -> {
    Comparator tmp = arg;      // error: name already defined
    return tmp.compare(rhs,lhs);
  };
  ...
}
```

When the first lambda declares the name `tmp` then there is no collision because the enclosing method has not yet declared the name `tmp`. When the second lambda declares the name `tmp` then the compiler complains because at that point in the program text there is a variable named `tmp` declared in the enclosing method that is in conflict with the lambda's variable of the same name.

The Meaning of this and super in Lambda Expressions

As explained before, lambda expressions are expressions and a name used inside a lambda expression has the same meaning as in the enclosing scope. This is also true for the keywords `this` and `super`.

Example of the meaning of `this` in a lambda expression:

```
class Sequence {
  public void reverse_sort(Comparator arg) {
    System.out.println(this.toString());
                                // this denotes the sequence
    Comparator cmp = (Object lhs, Object rhs)  -> {
       System.out.println(this.toString());
                                // this denotes the sequence
       return arg.compare(rhs,lhs);
    };
  }
}
```

In the example, the comparator is provided as a lambda expression. The `this` keyword has the same meaning inside and outside the lambda expression; in both cases it refers to the sequence.

Example of the meaning of `this` in an anonymous inner class:

```
class Sequence {
  public void reverse_sort(Comparator arg) {
    System.out.println(this.toString());
                                // this denotes the sequence
    Comparator cmp = new Comparator() {
      public int compare(Object lhs, Object rhs)  {
        System.out.println(this.toString());
                               // this denotes the comparator
        System.out.println(Sequence.this.toString());
                                // this denotes the sequence
        return arg.compare(rhs,lhs);
      }
    };
  }
}
```

In the example, the comparator is provided as an anonymous inner class. When the `this` keyword is used inside the inner class it now refers to the comparator

instead of the sequence. In order to refer to the sequence from with the inner class the qualified name `Sequence.this` must be used.

In analogy, the keyword `super` refers to the supertype of the enclosing type when used in a lambda expression and to the supertype of the anonymous class when used in an anonymous class.

Example of the meaning of `super` in a lambda expression:

```
class Sequence implements Cloneable {
  public Sequence clone() {
    Sequence copy = null;
    Supplier<Sequence> cloner;
    cloner = () -> {
      try {
        return (Sequence)super.clone();
                    // super refers to supertype of Sequence
      } catch(CloneNotSupportedException e) {
        throw new InternalError(e);
      }
    };
    copy = cloner.get();
    ....
    return copy;
  }
}
```

Example of the meaning of `super` in an anonymous inner class:

```
class Sequence implements Cloneable {
  public Sequence clone() {
    Sequence copy = null;
    Supplier<Sequence> cloner;
    cloner = new  Supplier<Sequence>() {
      public Sequence get() {
        try {
          // error: ClassCastException
          // super refers to supertype of anonymous Supplier
          return (Sequence)super.clone();
        } catch(CloneNotSupportedException e) {
          throw new InternalError(e);
        }
      }
    };
    copy = cloner.get();
    ....
    return copy;
  }
```

```
}
```

When we need to refer to the supertype of the enclosing type a qualification is needed, i.e., we must use `Sequence.super` instead of just `super` inside the anonymous class.

Example, same as before with error eliminated:

```
class Sequence implements Cloneable {
  public Sequence clone() {
    Sequence copy = null;
    Supplier<Sequence> cloner;
    cloner = new  Supplier<Sequence>() {
      public Sequence get() {
        try {
          return (Sequence)Sequence.super.clone();  // fine
        } catch(CloneNotSupportedException e) {
          throw new InternalError(e);
        }
      }
    };
    copy = cloner.get();
    ....
    return copy;
  }
}
```

Binding Restricted to Implicitly Final Variables

A lambda expression can refer to variables of its enclosing scope. This binding of a name used in the lambda to a local variable of the enclosing scope requires that the respective variable is not modified. In order to prevent modification, the outer variable can either be explicitly declared as `final`, or the compiler implicitly treats is as effectively `final`. Why is this restriction to `final` variables? Why can't a lambda refer to a non-`final` local variable of the enclosing scope?

The reason is that bindings to local variables collide with the intuitive understanding of local variables. Of a variable that is local to a method we expect two things:

- The local variable has a lifetime from where it is defined until the end of the method, and
- local variables are invisible to other threads and therefore will never be subject to race conditions and are inherently thread-safe.

Both properties vanish as soon as a lambda expression has a binding to a local variable in the enclosing method. The lambda expression may outlive the termination of the method, for instance if it is returned from the method or assigned to a field. Along with the surviving lambda expression all bound local variables would stay around long after exit from the method. In other words, a local variable's lifetime would no longer be tied to the local context in which it was defined.

Furthermore, the local variable might be accessed concurrently, if for instance the local variable is bound to a lambda that is passed to another thread and executed concurrently. This would introduce an entirely new category of race conditions, namely race conditions for local variables - which traditionally have been thread-safe because they are created on the stack and for this reason inaccessible to other threads. Detecting and correctly handling race conditions is error-prone to begin with and adding even more opportunities for race conditions adds to the complexity of multi-threading.

The loss of the local variables' two expected properties - limited lifetime and thread-safety - is acceptable if the local variables are immutable. If a variable is never modified then there is no potential for race conditions; it simply does not matter where and when the immutable value is read or whether this read access happens sequentially or concurrently. For this reason, the binding to local variable of the enclosing method is restricted to immutable, i.e., `final` variables.

For illustration, let us study an example.

Example #1: Binding to a Local Variable of a Primitive Type

Example of a lambda expression with a binding to local variables of the enclosing method:

```
Runnable[] makeTasks() {
  int cnt = 0;

  Runnable incrTask = () -> {
    while (true) {
       cnt++; // error: variable must be effectively final
    }
  };

  Runnable watchTask  = () -> {
    do {
        // nothing
    }  while(cnt < 100_000 );
    System.out.println(Thread.currentThread().getName()
                       +": stops at "+cnt);
  };
```

```
    return new Runnable[] {watchTask,incrTask};
}
```

The local variable `cnt` is declared inside method `makeTasks`. The variable `cnt` is used in the two lambda expressions that appear in the method. Both lambdas implement the `Runnable` interface and will be executed asynchronously in separate threads.

```
class Test {

    public static void main(String... args) {
        Runnable[] tasks =  makeTasks();

        watchdog[i] = new Thread(tasks[0],"watchdog"i);
        incrementer[i]
                    = new Thread(tasks[1],"incrementer"i);
        incrementer[i].setDaemon(true);

        watchdog[i].start();
        incrementer[i].start();
    }
}
```

If the lambdas run in different threads, there is concurrent access to the integral value `cnt` declared locally in the `makeTasks` method. One of the accesses is a modification, namely the attempted increment that the compiler rejects as an error. Concurrent access to mutable data is error-prone. The example, for instance, has a visibility problem. The watchdog thread might see a stale value of `cnt`, e.g. the initial value `0`, although the incrementer thread keeps incrementing the `cnt` until overflow and beyond. Without proper precautions this tiny program is incorrect.

Fortunately, the compiler refuses to compile the incorrect program. The `cnt` variable must be effectively `final` and the compiler flags the attempted increment as an error. If the `cnt` variable were indeed (effectively) `final`, then all access in all lambdas would be non-mutating, which is thread-safe and does not have visibility problems. Essentially, the restriction to (effectively) final variables is a precaution in order to prevent errors and to ease use of lambda expressions and name binding.

Example #2: Binding to a Local Variable of a Reference Type

In the case study above the local variable in question was of a primitive type, namely an `int` variable. What happens if the local variable is of a reference type?

References, too, must be effectively `final` if lambda expressions want to bind to them. The required explicit or implicit `final` declaration on a reference only affects the reference variable itself, but not the referenced object. That is, we must not modify the bound reference variable, but we may modify the referenced object. This perfectly makes sense.

The reference variable itself is placed on the stack of the method in which it is local. Its lifetime is tied to the method. This is exactly the same as for the primitive type local variable.

The referenced object, in contrast, is allocated on the heap and its lifetime is independent of the method and instead tied to the fact whether the object is reachable or unreachable. We also know that we are responsible to ensure thread-safety if we make an object accessible to multiple threads.

The previously studied example could be re-written using a local reference variable instead of a primitive type variable. We need a reference to a thread-safe counter and for this purpose we use a `LongAdder` from package `java.util.concurrent.atomic`.

Example of lambda expressions with bindings to a local reference variable of the enclosing method:

```
Runnable[] makeTasks() {
  final LongAdder cnt = new LongAdder();

  Runnable incrTask = () -> {
    while (true) {
       cnt.increment();
    }
  };

  Runnable watchTask  = () -> {
    do {
        // nothing
    }  while(cnt.intValue() < 100_000 );
    System.out.println(Thread.currentThread().getName()
                       +": stops at "+cnt.intValue());
  };

  return new Runnable[] {watchTask,incrTask};
}
```

In this version the compiler no longer issues error messages, because the reference variable `cnt` to which the lambda expression binds is `final` - as is

required. The referenced `LongAdder` object is thread-safe and it is expressly designed for efficient, concurrent modification.

Example #3: Binding to a Local Variable of an Array Type

The fact that `final` references only prevent modification of the reference itself, but permit modification of the referenced object allows for bugs and errors. The example above is correct, because the reference refers to a thread-safe `LongAdder` object. Using references it is easy to make mistakes and inadvertently use a reference to a thread-unsafe object. Below is a version that demonstrates the potential pitfall.

Incorrect example of lambda expressions with bindings to local reference variables of the enclosing method:

```
Runnable[] makeTasks() {
  final int[] cnt = new int[] {0};

  Runnable incrTask = () -> {
    while (true) {
       cnt[0]++;
    }
  };

  Runnable watchTask  = () -> {
    do {
        // nothing
    }  while(cnt[0] < 100_000 );
    System.out.println(Thread.currentThread().getName()
                       +": stops at "+cnt[0]);
  };

  return new Runnable[] {watchTask,incrTask};
}
```

The counter is now a reference to an `int`-array of size 1 whose one and only entry contains the actual count value. Since we are using a `final` reference to the array, the compiler does not complain. Yet the example is incorrect. It has serious visibility problems: there is no guarantee that the watchdog thread will see the modifications produced by the incrementer thread.

The Array Boxing Hack

By the way, the above demonstrated approach is known as the so-called *array boxing hack*. It might occasionally be useful if you need to work around the "effectively final" restriction, which might be acceptable in single-threaded

situations. Still, it is error-prone. Don't use the array boxing hack unless you fully understand the implications of what you are doing.

Effectively Final Variables in Inner Classes

Note that the lambda expressions' restricted binding to effectively `final` variables of the enclosing scope is in line with the behaviour of anonymous and local inner classes. Anonymous and local inner classes also may have bindings to local variables of the enclosing scope. Traditionally, the binding was permitted only to explicitly `final` variables of the enclosing scope; since Java 8 we can omit the explicit `final` declaration and the compiler treats the bound variables as effectively `final`. The reasoning for the restriction to effectively `final` local variables is exactly the same as for lambda expressions.

No Restrictions for Bindings to Fields

In the previous section we discussed the binding of a name used in a lambda to a locale variable of the lambda's enclosing scope. A lambda expression can refer to other kinds of entities from enclosing scopes, e.g. a lambda expression may refer to a field defined in the enclosing class. The binding to a field of the enclosing class is not restricted to `final` or effectively `final` fields.

Here is an example of a lambda expression with bindings to its enclosing class's fields:

```
class UnrestrictedAccessToFields {
  private static int staticField = 0;
  private        int nonStaticField = 0;

  public void demonstrate() {
    int localVariable = 0;
    new Thread(()-> {
      staticField++;    // fine
      nonStaticField--; // fine
      localVariable++;  // error: local variable must be final
    }).start();
  }
}
```

Inside the lambda expression the two fields are modified, which illustrates that the fields are neither `final` nor effectively `final`.

Fields and local variables are treated differently because they have different lifetime. As explained earlier (see section "Binding Restricted to Implicitly Final Variables") the reason for the "effectively `final`" restriction is the local variables' short lifetime.

The lifetime of a local variable is tied to the execution of the method in which it is defined, whereas the lambda itself might live longer. It must be prevented that a lambda expression defined in a method modifies local variables. For this reason, it can only bind to effectively `final` local variables.

The lifetime of a class's field, in contrast, exceeds the execution of the class's methods. There is no need to prevent that a lambda defined in a method modifies the class's fields. Hence the lambda can also bind to non-`final` fields and modify them.

Wrap-up

Names declared and used in a lambda expression are treated differently from names declared in a method or local or anonymous inner class.

- *Declared names.*

  A name declared in a lambda expression contributes to the lambda's enclosing context and *collides* with the same name declared in the lambda's enclosing context.

In contrast, a name declared in a method or local or anonymous inner class is an entity of its own and *shadows* corresponding names in the enclosing scope.

Used, but not declared names.

> A name used, but not declared in a lambda expression has the same meaning that it has in the lambda's enclosing context. The binding to *local variables* of the enclosing scope is only permitted for *effectively final* variables. The binding to *fields* of the enclosing class is *not restricted* to final fields.
>
> The same holds for a name used, but not declared in a method or local or anonymous inner class.

this and super.

> In a method or lambda expression `this` and `super` refer to an instance of the *enclosing* class type (or its superclass part).
>
> In contrast, in local or anonymous inner classes `this` and `super` refer to an instance of the *inner* class type (or its superclass part).

The Meaning of Jumps and Exits

In the previous section we learnt that names in a lambda expression have the same meaning as they have in the enclosing context. This bears the question whether keywords such as `break`, `continue`, `return`, and `throws` also have the same meaning that they would have in the enclosing context. This, however, is not true.

Keywords such as `break`, `continue`, `return`, and `throw` have the same meaning in a lambda expression that they have in a method. For instance, a `return` statement in a method triggers exit from the method. So does a `return` statement in a lambda expression: it triggers exit from the lambda expression. Keywords such as `break`, `continue`, `return`, and `throw` affect only the lambda expression, but never the enclosing method.

It was discussed during the design of lambdas whether so-called *non-local jumps* should be allowed, but eventually it was decided that there will be no support for non-local jumps in Java 8.

Local vs. Non-Local Jumps

In Java, the keyword `break` is only allowed in loops or `switch` statements and the keyword `continue` can only be used in loops. They permit jumps out of the loop or `switch` statement (`break`) or to the end of the loop body (`continue`). These rules hold for both methods and lambda expression. There is no difference.

For instance, if a lambda expression's body contains a loop, then we can break out of the loop using the `break` statement.

```
Consumer<int[]> reader = array -> {
  for (int i : array) {
    if (i < 0)
       break;                                          // fine
    ...
  }
};
```

Non-local jumps are not supported in Java, i.e. a `break` or `continue` statement in a lambda expression does not affect the lambda's enclosing context.

Example of an illegal `break` statement in a lambda expression:

```
void test(String... args) {
  IntConsumer reporter = i -> {
    System.err.println("illegal argument size "+i);
    break;            // error: break outside switch or loop
  };

  for (String s : args)  {
```

```
    if (s.length() == 0) {
       reporter.accept(0);
    }
    ...
  }
}
```

A `break` statement is only allowed in a loop or `switch` statement. Since the lambda body does not contain a loop, the compiler issues an error message. As a result there is no way that a jump statement in a lambda expression has any effect on the control flow of its enclosing method.

The example above can be fixed by placing the `break` statement in the enclosing method rather than the lambda.

Example, same as above, but non-local jump eliminated:

```
void test(String... args) {
  IntConsumer reporter = i -> {
    System.err.println("illegal argument size "+i);
  };
  for (String s : args)  {
    if (s.length() == 0) {
       reporter.accept(0);
       break;                                          // fine
    }
    ...
  }
}
```

Now, the `break` statement appears in the loop of the enclosing method, which is permitted and fine.

Return and Throw Statements in Lambda Expressions

The statements `return` and `throw` terminate the lambda expression in which they appear, but never cause exit from the enclosing method.

Example of `return` in a lambda expression:

```
void test(String... args) {
  IntConsumer reporter = i -> {
    System.err.println("illegal argument size "+i);
    return;                          // terminates the lambda
  };

  for (String s : args)  {
    if (s.length() == 0) {
       reporter.accept(0);
    }
    ...
  }
}
```

A `return` statement in a lambda expression triggers exit from the lambda body. It does not mean that the enclosing method terminates. In the example above, the `return` statement would end the lambda body, but not the `test` method.

If we want to end the `test` method we need to add a `return` statement to the method rather than the lambda.

Example, same as above, but with `return` in enclosing method:

```
void test(String... args) {
  IntConsumer reporter = i -> {
    System.err.println("illegal argument size "+i);
  };
  for (String s : args)  {
    if (s.length() == 0) {
       reporter.accept(0);
       return;                // terminates the test method
    }
    ...
  }
}
```

Now, the `return` statement appears in the enclosing method and terminates it.

The lambda expression can, of course, contain `return` statements for termination of the lambda body and/or passing back return values. The use of the `return` statement in a lambda expression is exactly the same as in methods.

```
Function<String,String> extractor = s -> {
  if (s.charAt(0) == '+' || s.charAt(0) == '-')
    return s.substring(1, s.length());
  else
    return s;
};
```

If there are multiple `return` statements in a lambda body, the types of the returned values may differ. The compiler looks for a common supertype of all return values and deduces the common supertype as the lambda expression's return type.

The rules `throw` statements are similar. A `throw` statement terminates the lambda expression (with an exception instead of a return value) and does not have any immediate effect on the enclosing method. If the enclosing method does not catch the exception thrown by the lambda, then the enclosing method also terminates with an exception. But this is due to the rules for exception propagation and handling and has nothing to do with lambdas. The effect of a `throw` statement in a lambda expression is exactly the same as in a method.

Recursive Lambda Expressions

Occasionally, one might need a function that recursively invokes itself. This can easily be achieved by means of anonymous inner classes. Here is an example of such a recursive function:

```
File[] findFiles(String dirname) {
  final File myDir = new File(dirname);

  if (myDir.isDirectory()) {
    final List<File> files = new ArrayList<>();
    final FileFilter filter = new FileFilter() {
                public boolean accept(File f) {
                    if (f.isDirectory()) {
                        files.addAll(Arrays.asList
                                    (f.listFiles(this)));
                        return false;
                    } else {
                        return f.isFile();
                    }
                }
    };
    files.addAll(Arrays.asList(myDir.listFiles(filter)));
    return files.toArray(new File[files.size()]);
  } else {
    return null;
  }
}
```

In this example, the `listFiles` method is invoked for a directory. The file filter passed to the `listFiles` method returns `true` for each file in the directory and `false` for each directory in the directory. It addition, for each directory in the directory the `listFiles` method is invoked with the same file filter. This way, all files in all directories are recursively collected in a list and eventually returned.

This is a situation where the file filter uses itself recursively in its own implementation. This is possible because the file filter is provided as an instance of an anonymous inner class. In the body of the class's `accept` method the file filter refers to itself via the `this` keyword. The file filter does so in order to pass itself to the `listFiles` method as the required file filter.

No such recursive use is possible with lambda expressions. Here is an attempt of such a recursive lambda expression. Note that it does not compile.

```
File[] findFiles(String dirname) {
  final File myDir = new File(dirname);

  if (myDir.isDirectory()) {
     final List<File> files = new ArrayList<>();
     final FileFilter filter = (File f) -> {
          if (f.isDirectory()) {
```

```
                files.addAll(Arrays.asList(f.listFiles(filter)));
                                // error: not yet initialized
                return false;
            } else {
                return f.isFile();
            }
        };
        files.addAll(Arrays.asList(myDir.listFiles(filter)));
        return files.toArray(new File[files.size()]);
    } else {
        return null;
    }
}
```

Due to lexical scoping, we cannot refer to the lambda expression inside the lambda expression's body via the `this` keyword. The `this` keyword does not refer to the lambda expression, but is the `this` reference of the enclosing context.

Our only chance to refer to the lambda inside the lambda is via a named variable. In this case we attempt to use the `filter` variable for this purpose. The `filter` variable is a local variable and all local variables must be initialized before their first use. When we use the `filter` variable in the definition of the lambda expression, the compiler rejects this use because the `filter` variable has not yet been initialized.

It means that recursive use of lambda expressions is not supported and anonymous inner classes must be used instead.

Generic Lambda Expressions Not Permitted

Generic lambda expressions are not permitted in Java and they are a corner case. This section explains under which circumstances you might miss the feature and how you can cope. It is likely that you will never need a generic lambda expression. Feel free to skip the section if you are not interested in exotic, rarely encountered issues.

When would we need a generic lambda expression?

Consider a functional interface whose single abstract method is generic: If you wanted to provide an implementation for the abstract method by means of a lambda expression the lambda expression would have to be generic, too. As already mentioned, no such thing as a generic lambda expression exists. There is no syntax for specifying type parameters for a lambda expression.

Let us explore an example of a functional interface with a generic abstract method:

```
interface Factory {
  <T> Generic<T> make();
```

```
}
```

It uses a generic class `Generic`:

```
class Generic<X> {
  ...
}
```

The following lambda expressions are illegal:

```
Factory f1 = ()->new Generic<>();      // error: lambda is not generic
Factory f2 = ()->new Generic<?>();     // error: illegal construction
Factory f3 = ()->new Generic<Long>();// error: lambda is not generic
```

The lambda expressions are illegal because they all boil down to a non-generic method that takes no arguments, does not throw exceptions, and returns an object of a parameterization of the generic class `Generic`. The left-hand side of the assignment, in contrast, requires a function that is generic and takes no arguments, does not throw, and returns a parameterization of `Generic`. The lambdas expressions almost match - except that they do not have a type parameter.

You might want to try something like this in order to specify a type parameter for the lambda expression, but the syntax does not exist:

```
Factory f4 = <T>()->new Generic<T>();// error: illegal syntax
```

Ultimately, there is no way to specify type parameters for lambda expressions. Lambda expressions are always non-generic.

How do we cope with the lack of generic lambda expressions?

Instead of a lambda expression we can use a method reference or an anonymous inner class.

Here is a solution using an anonymous inner class:

```
Factory f5 = new Factory() {
  public <T> Generic<T> make() { return new Generic<T>(); }
};
```

Implementing the functional interface by means of an anonymous inner class is no problem at all. Just define the required `make` method as a generic method.

Here is a solution using a constructor reference:

```
Factory f6 = Generic::new;
```

`Generic::new` `Generic`

Method and Constructor References".

Method and Constructor References

Along with lambda expressions, *method references* and *constructor references* were added to the language in Java 8. A method or constructor reference refers to a method or constructor without invoking it. They are a syntactic shortcut for creating a lambda expression out of an existing method/constructor.

Consider a lambda expression that is used as file filter in the `listFiles` method of class `java.io.File`:

```
File[] files = myDir.listFiles(
   (File f) -> { return f.isFile(); }
);
```

It takes a reference to a `File` object, calls its `isFile` method, and returns the method's result.

With a method reference we can simply say: use the `isFile` method. It looks like this:

```
File[] files = myDir.listFiles( File::isFile );
```

In the example `File::isFile` denotes a reference to a non-static method. There are also references to static methods (e.g. `System::gc`) and reference to constructors (e.g. `ArrayList::new`).

Method and constructor references consist of a qualifier, the `::` symbol, and a method name. Let us begin with a simplified version of the syntax of method and constructor references. The actual syntax production in the Java language specification is more complex and detailed: We will address some of the details later.

Here is a simplified version of the syntax:

```
ConstructorReference:
 TypeName       '::' 'new'

MethodReference:
 Expression '::' Identifier
 TypeName   '::' Identifier
```

For a constructor reference the qualifier is the name of a type. The `new` keyword serves as the name of the referenced constructor. The qualifier type must allow instance creation, e.g. it cannot be the name of an abstract class or an interface, because no objects of an abstract class or interface type can be created.

For a method reference the qualifier can be a type or an expression. The identifier is the name of the referenced method. An expression can only be used as the qualifier for non-static methods; the expression then is the object on which the method is invoked. Static methods are always referenced with a type name as the qualifier.

In the following we take a closer look at references to the various types of methods:

- constructors
- static methods
- non-static methods

Reference to Constructor

References to constructors have the form:

```
ConstructorReference:
 ClassType       '::'       NonWildTypeArgumentsopt       'new'
 ArrayType '::' 'new'
```

where `new` is the name of the constructor and the qualifier is either a `ClassType` or an `ArrayType`. The class type must be a type that permits creation of instances. It can not be the name of an abstract class, an interface, or of an enumeration type. All of these types do not permit creation of objects.

The optional `NonWildTypeArguments` are for the explicit specification of the type arguments of a generic constructor. This is needed for the rare cases in which the compiler cannot automatically infer the type arguments.

Here are examples of references to constructors:

reference to constructor	**equivalent lambda expression**

`String::new`	`() -> new String()`
	or
	`(String s) -> new String(s)`
	or
	`(value, offset, count)`
	`-> new String`
	`(value,offset,count)`

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

`String[]::new`	`size -> new String[size]`
`int[]::new`	`(int size)` `-> new int[size]`
`Outer.StaticInner::new`	`()` `-> new Outer.StaticInner()`

`Outer.NonStaticInner::new`	`(Outer outer)`
	`-> outer.new`
	`NonStaticInner()`

`Tuple<Number>::<Long>new`	`(Tuple<Long> pair)` `-> new Tuple<Number>(pair)`

Here is an example that illustrates the use of constructor reference. We have stream of floating point values and want to store the values in a list.

```
Stream<Double> doubles = …
Collector<Double,?,LinkedList<Double>> listCollector
    = Collectors.toCollection(LinkedList<Double>::new);
LinkedList<Double> list = doubles.collect(listCollector);
```

The example uses abstractions from the JDK package `java.util.stream`. The `Stream` interface describes a sequence of elements; in our example the element are floating point values of type `Double`. The `Stream` interface has a `collect` method that stores the stream's elements in a data store. A data store can for instance be a collection; in our example we want to use a `LinkedList<Double>` as the data store. The `collect` method needs a collector that knows how to create the data store. For creating collectors there is a helper class named `Collectors`; it has a `toCollection` factory method. It needs a `Supplier` that eventually provides the collection. In our example the supplier is the constructor of class `LinkedList<Double>`., denoted via the constructor reference `LinkedList<Double>::new`.

There is no support for specifying a particular signature to be matched, like for instance `String::new()`, `String::new(String)`, `String::new(StringBuilder)`, `String::new(StringBuffer)`, or `String::new(char[], int, int)`. When the constructor is overloaded, i.e. if there is more than one constructor, the appropriate constructor is selected based on the type inference context.

Here is an example that refers to two overloaded constructors of class String, namely `String(StringBuilder)` and `String(StringBuffer)`:

```
Stream<StringBuilder> builders = …
Stream<String> strings = builders.map(String::new);
```

```
Stream<StringBuffer> buffers = …
Stream<String> strings = buffers.map(String::new);
```

The compiler picks the right constructor depending on the context in which the constructor reference appears.

Similarly, the meaning of a reference to a generic constructor depends on the context. Consider the following generic class with a generic constructor:

```
class Tuple<A> {
    private A fst, snd;
    public <X extends A> Tuple(Tuple<X> other) {
        fst = other.fst;
        snd = other.snd;
    }
}
```

The constructor reference `Tuple::new` can refer to various parameterizations such as `Tuple<String>::<String>new`, `Tuple<Number>::<Long>new`, or `Tuple<Object>::<Date>new`. The compiler selects the appropriate parameterization for each given context.

Here are some examples of a context in which the constructor reference `Tuple::new` appears:

```
Function<Tuple<String>,Tuple<String>> ctorRef
    = Tuple::new;          // refers to Tuple<String>::<String>new

Function<Tuple<Long>,Tuple<Number>>   ctorRef
    = Tuple::new;          // refers to Tuple<Number>::<Long>new

Function<Tuple<Date>,Tuple<Object>>   ctorRef
    = Tuple::new;          // refers to Tuple<Object>::<Date>new
```

where `Function` is the functional interface `java.util.function.Function`:

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

Here is an example that uses references to different parameterizations of the generic `Tuple` constructor:

```
Stream<Tuple<Number>> numbers = …
Stream<Tuple<String>> strings = …
Stream<Tuple<Object>> objects =
Stream.concat(numbers.map(Tuple::new),strings.map(Tuple::new));
```

The `concat` method of interface `Stream` concates two sequences. It requires two arguments, both of which must be streams of the exact same type. In the example above we have two streams of different type, namely a `Stream<Tuple<Number>>` and a `Stream<Tuple<String>>` and we intend to concatenate them to a `Stream<Tuple<Object>>`. In order to produce a stream of object tuples the `concat` method needs two streams of object tuples and we must convert the two input streams to the required type before we can pass them to the `concat` method. We achieve the conversion by mapping the number and string tuples to object tuples; as a mapping functions we use the constructors `Tuple<Object>::<Number>new` and `Tuple<Object>:: <String>new`. As the context provides enough information for type inference, we can denote both constructors as `Tuple::new`; the compiler does the rest and automatically infers the omitted type parameters.

A detailed discussion of the process of selecting an appropriate signature (in case of overloading) or an appropriate parameterization (in case of generics) is given later in the section on "Target Typing".

Reference to Static Method

References to static methods have the form:

```
MethodReference:
 ReferenceType '::' NonWildTypeArgumentsopt Identifier
```

where `Identifier` is the name of the static method and `ReferenceType` is the name of method's declaring type. The optional `NonWildTypeArguments` are for the explicit specification of the type arguments of a generic method. This is needed for the rare cases in which the compiler cannot automatically infer the type arguments.

Here are examples of references to static methods:

reference to static method	equivalent lambda expression
`String::format`	`(String fmt, Object... args)` `-> String.format(fmt,args)`

`System::currentTimeMillis`	`()`
	`-> System`
	`.currentTimeMillis()`

`Arrays::toString`	`Array` `-> Arrays.toString(array)`

`Arrays::asList`	`Array` `  -> Arrays.asList(array)`

`Arrays::<String>asList`	`(String[] array)`
	`  -> Arrays.asList(array)`
	or
	`Array`
	`  -> Arrays`
	`  .<String>asList(array)`

`TimeUnit::values`	`() -> TimeUnit.values()`
`Enum::valueOf`	`(type,name)` `  -> Enum.valueOf(type,name)`

`TimeUnit::valueOf`	`(type,name)`
	`  -> TimeUnit.valueOf`
	`  (type,name)`
	or
	`(Class<TimeUnit> type,`
	`  String name)`
	`-> { return TimeUnit`
	`  .valueOf(type,name);`
	`  }`

`T::valueOf` (for type parameter `T extends Enum<T>`)	`(Class<T> t, String s)` `  -> T.valueOf(t, s)`

`Collections::sort`	`List`
	`  -> Collections.sort(list)`
	or
	`(list ,comparator)`
	`  -> Collections.sort`
	`  (list,comparator)`

Here is an example that uses references to static methods:

```
String s = DoubleStream.of(1, 2, 3)
                .map(Math::log)
                .mapToObj(Double::toString)
```

```
                .collect(Collectors.joining(",  "));
```

It produces the following string:

```
0.0,  0.6931471805599453,  1.0986122886681098
```

We create a stream of three floating point values, map them to their logarithms using the static `log` method from class `Math`, map the logarithms to strings using the static `toString` method from class `Double`, and eventually concatenate the strings to a single string using a string collector with delimiters.

When a static method is overloaded, i.e. if there is more than one method with the same name, the appropriate method is selected based on the type inference context. For instance, class `Collections` has two static methods named `sort` one that takes a list and another one that takes a list plus a comparator. Which one is referenced via the method reference `Collections::sort` depends on the context.

Similarly, the meaning of a reference to a generic method depends on the context. For example, the method reference `Arrays::asList` can refer to various parameterizations such as `Arrays::<String>asList`, `Arrays::<Date>asList`, or `Arrays::<Long>asList`. The compiler selects the appropriate parameterization for each given context.

Here is an example that uses a reference to a generic static method:

```
Map<Class<? extends Enum<?>>,List<Enum<?>>>
             getValueMap(Class<? extends Enum<?>>... enumTypes) {

  Map<Class<? extends Enum<?>>,List<Enum<?>>> values
    = new HashMap<>();

  for (Class<? extends Enum<?>> enumType : enumTypes)   {
    Function<Class<? extends Enum<?>>,Enum<?>[]> getEnumConstants
      = c -> c.getEnumConstants();
    Function<Enum<?>[],List<Enum<?>>> convertToList
      = Arrays::asList;
    values.computeIfAbsent(enumType,
                        getEnumConstants.andThen(convertToList));
  }
  return values;
}
```

In this example we create a map that associates an enum type with the list of the enum constants for the respective enum type. The map is populated using the `computeIfAbsent` method from interface `Map`. It takes a mapping function that computes the associated values for a given key. In our example the mapping function must compute a list of enum constants for each enum type. We provide the mapper function by composing two functions: the first function

(`getEnumConstants`) retrieves all enum constants for the enum type and returns them as an array; the second function (`convertToList`) turns the array into a list. The composition is achieved via the `andThen` method of interface `Function`. The second function in the composition is denoted by a reference to the static generic method `asList` from class `Arrays`. Its type parameter, which would be `Enum<?>` in our example, is automatically deduced by the compiler.

Reference to Non-Static Method

If we want to invoke a non-static method we need an object on which the non-static method can be invoked. This target object is the so-called *receiver*. We can provide the receiver explicitly (as an expression) or we can supply it implicitly, i.e., later when the method is invoked.

Accordingly, references to non-static methods have the form:

```
MethodReference:
 Expression          '::'  NonWildTypeArgumentsopt  Identifier
 ReferenceType  '::' NonWildTypeArgumentsopt Identifier
```

where `Identifier` is the name of the non-static method and `ReferenceType` is the name of method's declaring type. Details will be discussed in the subsequent sections; we will discuss which of the two forms (with or without a receiver) is best used in which situation.

The optional `NonWildTypeArguments` are for the explicit specification of the type arguments of a generic method. This is needed for the rare cases in which the compiler cannot automatically infer the type arguments.

Unbound Receiver

An example of a reference to a non-static method with an unbound receiver is `String::length`. The `length` method as such is a non-static method defined in class `String`; it takes no argument and returns an `int` value.

The method itself is not to be confused with the method reference `String::length`. A reference to a non-static method always needs a receiver object. The receiver object in the example of `String::length` is a `String` object that is used when the `length` method is invoked via the method reference. Obviously, the method reference `String::length` does not specifiy any particular string object as the receiver. This is why we talk of an *unbound receiver*.

If the receiver is not specified as part of the method reference, it must be supplied later when the method is called. As a consequence the method reference `String::length` does not denote a method that takes no argument and returns an `int` value, as one might believe when looking at the `length` method's

signature. Instead the method reference `String::length` denotes a method that takes one argument of type `String`, namely the receiver object, and returns an `int` value. In other words, the method reference `String::length` has the signature `(String) -> int` because it is equivalent to the lambda expression `(String s) -> { return s.length(); }`.

This might be slightly confusing at first sight. So, take a mental note of the fact that method references to non-static method with an unbound receiver always take an additional first argument, namely the receiver.

Here is an example that illustrates the use of the method reference `String::length`:

```
static double averageStringLength(String... strings) {
  return Arrays.stream(strings)
               .mapToInt(String::length)
               .average()
               .getAsDouble();
}
```

For a stream of strings its elements (the strings) are mapped to integer values (their string length), the average is calculated and returned. As a mapper the method reference `String::length` is used, which demonstrates that `String::length` is a function that takes a `String` and returns an int.

Here are further examples of references to non-static methods with an unbound receiver.

reference to unbound non-static method	**equivalent lambda expression**
`String::length`	`(String s) -> s.length()`
`List::equals`	`(lhs, rhs) -> lhs.equals(rhs)`
`List<Long>::equals`	`(List<Long> lhs, Object rhs) -> lhs.equals(rhs)`
`Logger::log`	`(Logger logg, Level sev, String msg) -> logg.log(sev,msg)`
`int[]::clone`	`(int[] a) -> a.clone()`

`Collection::toArray`	`c -> c.toArray()`
	or
	`(Collection<?> c) -> c.toArray()`
	or
	`(Collection<String> c) -> c.toArray()`

`Collection::toArray`
or
`Collection<Number>::toArray`
or
`Collection::<Long>toArray`
or
`Collection<Number>::<Long>toArray`
	`(Collection<Number> c, Long[] a) -> c.toArray(a)`

`T::ordinal` (for type parameter `T extends Enum<T>`)	`(T t) -> t.ordinal()`
`Outer.Inner::innerMethod`	`(Outer.Inner inner) -> inner.innerMethod()`

`Thread.UncaughtExceptionHandler::uncaughtException`	`(h, t, e) -> h.uncaughtException(t,e)`
	or
	`(Thread.UncaughtExceptionHandler h, Thread t, Throwable e) -> h.uncaughtException(t,e)`

Here is another example in which a reference to a non-static method with unbound receiver is used:

```
static <T> Class<?>[] whichTypes(T[] array) {
        return Arrays.stream(array)
                     .map(T::getClass)
                     .collect(Collectors.toSet())
                     .toArray(new Class<?>[0]);
}
```

All elements in a stream of objects of unknown type `T` are mapped to their types, i.e., their corresponding `Class` objects. The types are collected to a set, which is eventually converted to an array. As a mapper function the method reference `T::getClass` is used. As already pointed our earlier, the non-static method `getClass` takes no arguments, but the method reference `T::getClass` does take an argument. Since the method reference `T::getClass` does not specify the receiver object, it denotes a function that takes an object of unknown type `T` (the receiver) and returns its type (as a `Class` object).

Bound Receiver

In the previous section we used references to non-static methods where the receiver was not specified. Naturally, we can choose to provide a particular receiver object as part of the method reference. In this case we talk of a *bound receiver*.

An example of a reference to a non-static method with a bound receiver is `System.out::println`. If refers to the `println` method of class `PrintStream` and specifies that the `println` method will be applied to a particular receiver object, namely the standard output stream `System.out`. The method reference `System.out::println` has the signature `(Object)->void` and is equivalent to the lambda expressions `(Object o) -> System.out.println(o)`.

The method reference `System.out::println` is frequently used for debugging purposes like in the following code snippet:

```
static List<String> findStringIn(String match, String[] strings){
        return Arrays.stream(strings)
                .peek(System.out::println)
                .filter(s->match.equals(s))
                .peek(System.out::println)
                .collect(Collectors.toList());
}
```

A filter is applied to an array of string. Via the `peek` operation the strings are printed to `System.out` before and after filtering in order to check whether the filter has the expected effect.

Each time we want to specify the receiver explicitly, we must use an expression that refers to the receiver. In the example above the expression in question was `System.out`. More generally, the expression is of the form:

```
Expression:
 ExpressionName
 Primary
 'super'
 TypeName '.' 'super'
```

Here are further examples of references to non-static methods, where the receiver is explicitly specified:

reference to unbound non-static method	equivalent lambda expression

`Thread`
`.currentThread()`
`.getName()::length`
	`()`
	`->  Thread.currentThread()`
	`.getName().length()`

`"xyz"::length`	`() -> "xyz".length()`
`this::equals`	`other -> this.equals(other)`
`super::equals`	`other -> super.equals(other)`

`Logger`
`.getLogger("global")::log`
	`(Level sev, String msg)`
	`->  Logger.getLogger("global")`
	`.log(sev,msg`

`new int[] {1,2}::clone`	`() -> new int[] {1,2}.clone()`
`new Thread()::start`	`() -> new Thread().start()`

`new`
`Thread(Framework::test)`
`::start`
	`()`
	`->  new  Thread(`
	`()->Framework.test()`
	`).start()`

`Arrays.asList(1L,2L,3L)`
`::toArray`
or
`Arrays.asList(1,2,3)`
`::<Long>toArray`
	`a`
	`->  Arrays.asList(1L,2L,3L)`
	`.toArray(a)`
	or
	`a`
	`->  Arrays.asList(1,2,3)`
	`.<Long>toArray(a)`
	or
	`(Long[] a)`
	`->  Arrays.asList(1,2,3)`
	`.toArray(a)`

`Outer.this::outerMethod`	`() -> Outer.this.outerMethod()`
`Outer.super::hashCode`	`() -> Outer.super.hashCode()`

Let us consider another example of a reference to a non-static method with a bound receiver:

```
static List<String> findStringIn(String match, String[] strings){
        return Arrays.stream(strings)
                .filter(match::equals)
                .collect(Collectors.toList());
}
```

All strings from an input array (`strings`) are compared to a given string (`match`); all matching strings are collected in a result list, which is returned. As a filter the method reference `match::equals` is used. It refers to the `equals` method of class `String`; the `equals` method is called on the receiver object `match`. The method reference `match::equals` is equivalent to the lambda expression `(String s) -> { return match.equals(s); }`. The net effect of filtering with the predicate `match::equals` is that every string in the input stream is compared to the string object `match`.

Method References in Action

Here is a final, more complex example that shows method references in action. Feel free to skip it if you have seen enough of method references for now. The example illustrates how conveniently method references solve various problems, but it also points out certain limitations of method references.

In the example we try to gather the ids of all runnable threads in an application:

```
static Set<Long> findRunnableThreads() {
  Function<ThreadInfo,Thread.State>
  first      = ThreadInfo::getThreadState;
  Function<Thread.State,Boolean>
  second     = Thread.State.RUNNABLE::equals;
  Function<ThreadInfo,Boolean>
  isRunnableFunction = first.andThen(second);
  Predicate<ThreadInfo>
  isRunnablePredicate = isRunnableFunction::apply;

  return Arrays.stream(
    ManagementFactory.getThreadMXBean().dumpAllThreads(true,true)
  ).filter(isRunnablePredicate)
   .map(ThreadInfo::getThreadId)
   .collect(Collectors.toSet());
}
```

First, an array of `ThreadInfo` objects for all live threads is retrieved (via the `ThreadMxBean`'s `dumpAllThread` method). Through a filter (`isRunnablePredicate`) all runnable threads are selected, mapped to their thread ids, and the thread ids eventually stored in a set.

The predicate `isRunnablePredicate` takes a `ThreadInfo` object, retrieves the corresponding thread state (via `ThreadInfo::getThreadState`), compares the thread state to the runnable state (via `RUNNABLE::equals`), and returns the resulting `boolean` value. The predicate `isRunnablePredicate` is essentially composed (via `andThen`) from the two functions `ThreadInfo::getThreadState` and `RUNNABLE::equals`.

An additional complication stems from the fact that the result of combining two functions via the `andThen` method is again a function, and not a predicate. More precisely, the composed function returned from `andThen` is of type `Function<ThreadInfo, Boolean>`, while we need a predicate of type `Predicate<ThreadInfo>`, which we can pass it to the stream's `filter` operation. The only difference between a `Function<ThreadInfo, Boolean>` and a `Predicate<ThreadInfo>` is that the function has an `apply` method that returns a `Boolean` whereas the predicate has a `test` method that returns a `boolean`. The conversion problem is easily solved by passing the method reference `isRunnableFunction::apply` to the stream's filter operation instead of the `isRunnableFunction` itself. When the `apply` method is invoked, it returns a `Boolean` value, which is auto-unboxed to a `boolean` value, which matches the required return type of the predicate's `test` method - e voilà - the conversion problem is solved.

By the way, the entire example can equally well be expressed without method reference, for instance like this:

```
static Set<Long> findRunnableThreads() {
  Predicate<ThreadInfo> isRunnablePredicate
  = info -> info.getThreadState().equals(Thread.State.RUNNABLE);

  return Arrays.stream(
    ManagementFactory.getThreadMXBean().dumpAllThreads(true,true)
  ).filter(isRunnablePredicate)
   .map(ThreadInfo::getThreadId)
   .collect(Collectors.toSet());
}
```

You might wonder why the `isRunnablePrediate` is more compactly expressed via a lambda expression compared to the rather lengthy composition of method references we have seen earlier. In principle we can condense the combination of method reference to a more compact notation. Let us try it.

Original composition using method references:

```
Function<ThreadInfo,Thread.State>
first               = ThreadInfo::getThreadState;
Function<Thread.State,Boolean>
second              = Thread.State.RUNNABLE::equals;
Function<ThreadInfo,Boolean>
```

```
isRunnableFunction  = first.andThen(second);
Predicate<ThreadInfo>
isRunnablePredicate = isRunnableFunction::apply;
```

An attempted compaction:

```
Predicate<ThreadInfo> isRunnablePredicate
= ThreadInfo::getThreadState                              // error
 .andThen(Thread.State.RUNNABLE::equals)
 ::apply;
```

The elegant use of a lambda expression:

```
Predicate<ThreadInfo> isRunnablePredicate
= info -> info.getThreadState().equals(Thread.State.RUNNABLE);
```

Our attempted compaction is rejected by the compiler with the error message "method reference not expected here". This stems from the fact that a method invocation context is not permitted as a type inference context. The section on "Target Typing" discusses type inference in detail. For the time being, suffice to say that we would have to insert a cast in order to make it compile. Then it looks like this:

The attempted compaction, now fixed:

```
Predicate<ThreadInfo> isRunnablePredicate
= ((Function<ThreadInfo,Thread.State>)ThreadInfo::getThreadState)
 .andThen(Thread.State.RUNNABLE::equals)
 ::apply;
```

Either way, the solution using a lambda expression is probably the most readable solution.

Bottom Line: Method References vs. Lambda Expressions

Method references provide a compact and readable notation for functions that look more complex when denoted by an equivalent lambda expression. You have seen many such examples in the tables throughout this section. The key reason for their readability and compactness is that the compiler infers practically everything for a method references (details of type inference are covered in the section on "Target Typing").

On the other hand, a limitation of method references is that they are not expected in front of the method selection symbol `'.'` in method calls. This complicates their composition via operations such as `and`, `or`, `negate` (from interface `Predicate`), `compose`, `andThen` (from interface `Function`), and `chain` (from interface `Consumer`).

Functional Interfaces

Functional interfaces are a special category of interfaces. They are used in conjunction with the type deduction for lambda expressions and method/constructor references.

When lambda expressions and method/constructor references were added to the Java programming language, the language designers tried to keep matters simple and to avoid major modifications of the language and its type system. At the same time they had to add an entirely new concept to the language, namely the concept of "functions". Conceptually, both lambda expressions and method/constructor references express functions. The most natural approach for adding such a new concept would have been to extend Java's type system and invent a new category of types, namely "function types" that describes functions and their signatures. A function type could have looked like `(int)->void` for a function that takes `int` and returns `void`, for instance. The language designers decided against such a major addition to the language's type system. Instead, the looked for a way to integrate lambda expressions and method/constructor references into the language without inventing new types or type categories.

The solution they came up with are *functional interface types* and a type inference process called *target typing* that figures out a matching functional interface type for each lambda expression or method/constructor reference. Both functional interfaces and target typing did already exist in Java before lambda expressions and method/constructor references had been invented.

The term "functional interface" is just a fancy name for an interface with one abstract method. Interfaces with one method did exist in Java all along. Examples are `Runnable`, `Callable`, and `Comparable`.

The "target type", too, is a familiar concept in Java. When an expression appears in a context, its type must be compatible with a type expected in that context. The expected type is called the *target type*. For lambda expressions and method/constructor references the target type is inferred by the compiler and must be a functional interface type.

In the following we explore functional interfaces in further detail. The process of target type deduction is explained later in the section on "Target Typing".

Definition

A functional interface is an interface that has just one abstract method.[1] Many existing interfaces in the JDK have this property, e.g. `Runnable`, `FileFilter` and `ActionListener`. In conjunction with the extension of the collection framework many more functional interfaces were invented. Here are a couple of simple, yet typical examples.

Examples of simple functional interfaces (taken from the JDK source code):

```
public interface Runnable {
    public abstract void run();
}

public interface Callable<V> {
    V call() throws Exception;
}

public interface Comparable<T> {
    public int compareTo(T o);
}

public interface FileFilter {
    boolean accept(File pathname);
}

public interface AutoCloseable {
    void close() throws Exception;
}
```

Functional Interfaces with Additional Non-Abstract Methods

Functional interfaces must have exactly one *abstract* method. In addition, the interface can have an arbitrary number of *non-abstract* methods. These non-abstract methods can be default methods, methods inherited from class `Object`, or static methods.

Example of a functional interface with a method inherited from class `Object`:

```
@FunctionalInterface[2]
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
       // not abstract; implementation inherited from Object
```

---

[1] These interfaces were initially called *SAM (single abstract method) Types*.

[2] Note functional interfaces can be qualified by the `@FunctionalInterface` annotation. It is discussed in the section on "Annotation @FunctionalInterface":

```
}
```

The `compare` method is abstract, because it has no implementation. The `equals` method is non-abstract, although it has no implementation specified in the declaration of interface `Comparator`. But every class inherits a default implementation of the `equals` method from class `Object`. For this reason, the `equals` method is non-abstract and the `Comparator` interface is a functional interface.

All public methods inherited from class `Object` are thus considered non-abstract methods. This is different for protected methods inherited from class `Object`.

Example of a functional interface with a protected method inherited from class `Object`:

```
@FunctionalInterface
public interface Producer<T> extends Cloneable {
    T produce();
    Object clone();      // error; multiple abstract methods
}
```

The `clone` method is abstract, although it is inherited from class `Object`. But the `clone` method is `protected` in class `Object` and thus no implementation is publicly available in a subclass of `Object`. For this reason, the `clone` method is considered abstract and the `Producer` interface is not a functional interface.

Functional interfaces can have default static methods in addition to the single abstract method.

Example of a functional interfaces with additional default method:

```
@FunctionalInterface
public interface Function<T, R> {
  public R apply(T t);

  public default <V> Function<V, R> compose
                      (Function<? super V, ? extends T> before) {
    Objects.requireNonNull(before);
    return (V v) -> apply(before.apply(v));
  }
  public default <V> Function<T, V> andThen
                       (Function<? super R, ? extends V> after) {
    Objects.requireNonNull(after);
    return (T t) -> after.apply(apply(t));
  }
  public static <T> Function<T, T> identity() {
    return t -> t;
  }
}
```

The `apply` method is abstract, i.e., it has no implementation. It is the functional method of interface `Function`. All other methods have implementations. The methods `compose` and `andThen` are default methods[3]. The `identity` method is a static interface method[4]. Default methods and static methods cannot be abstract; they always have an implementation. The only abstract method is the `apply` method and for this reason the `Function` interface is a functional interface.

Annotation @FunctionalInterface

Interface definitions can be marked with the annotation `@FunctionalInterface`. The `@FunctionalInterface` annotation is defined in package `java.lang`.

Example of an interface with a `@FunctionalInterface` annotation:

```
@FunctionalInterface
public interface Readable {
    public int read(java.nio.CharBuffer cb) throws IOException;
}
```

`@FunctionalInterface` is an informative annotation that indicates that an interface is intended to be a functional interface. The compiler checks whether the annotated type is an interface and whether it has one abstract method. Otherwise it issues an error message.

The purpose is to ensure that a functional interface remains a functional interface and is not inadvertently turned into a regular interface, for instance, by addition of another abstract method.

Example of an error message triggered due to a `@FunctionalInterface` annotation:

```
@FunctionalInterface
private interface Producer<R>  {
                    // error: not a functional interface
  R produce();
  R produce(R arg);
}
```

Functional interfaces need not be qualified by the `@FunctionalInterface` annotation. The compiler treats any interface meeting the definition of a functional interface as a functional interface regardless of whether or not a `@FunctionalInterface` annotation is present on the interface declaration.

---

[3] Default methods are discussed in the section on "Default Interface Methods".

[4] Static interface methods are covered in the section on "Static Interface Methods".

Generic Functional Interfaces

Generic Interface

Functional interfaces may be generic. We have already seen several examples.

Examples of generic functional interfaces:

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}

@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}

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

Non-Generic Interface with a Generic Single Abstract Method

In these examples the functional interface is generic and the single abstract method uses the enclosing interface's type parameters. It is even allowed that the single abstract method has its own type parameters. Here is an example of a (contrived) non-generic functional interface with a generic method:

```
@FunctionalInterface
interface RecursiveExecutor {
  <T> T execute(RecursiveTask<T> a);
}
```

Note, that functional interfaces with a generic method cannot be implemented by means of lambda expressions, because generic lambda expressions are not permitted[5]. Only a method reference may be used as the implementation of the functional interface `RecursiveExecutor`, like in this example:

```
RecursiveExecutor pool = ForkJoinPool.commonPool()::invoke;
```

[5] See the sections on "Generic Lambda Expressions" and on "Type Inference " for details.

Raw Types and Parameterizations of Generic Interfaces

Consider a generic interface that is a functional interface like for instance `Callable<V>`. If we replace the type parameter `V` by a concrete type, then the resulting parameterization of the generic interface is functional, too. That is, `Callable<String>`, `Callable<Long>`, etc. are a functional interface as well.

The same holds if we drop the type parameters altogether and use the raw type. That is, the raw type `Callable` is a functional interface, too.

Below are examples of subtypes of parameterizations and raw forms of a generic functional interface: The comments denote the respective inherited single abstract method.

```
@FunctionalInterface
public interface StringCallable extends Callable<String> {
  //  String call() throws Exception;
}

@FunctionalInterface
public interface RawComparator extends Comparator {
  //  int compare(Object o1, Object o2);
}

@FunctionalInterface
public interface StringProducer<T> extends Function<T, String> {
  //  public String apply(T t);
}
```

Non-Functional Interfaces that Collapse into Functional Interfaces

In rare cases the parameterization of a non-functional generic interface can be a functional interface. This can happen if a generic interface has overloaded methods that collapse into a single abstract method for certain parameterizations. This will only happen infrequently, but here is an example of this corner case.

Examples of parameterization of a non-functional generic interface with collapsing methods:

```
interface Sink<T, N extends Number> {  // not functional
   void consume(T arg);
   void consume(N arg);
}
@FunctionalInterface
interface NumberSink extends Sink<Number, Number> {
  // void consume(Number arg);
}
```

The generic `Sink` interface has two abstract methods, both named `consume`, but with different argument list. If both type parameters are replaced with the same

concrete type the overloaded methods collapse into a single abstract method. For this reason the subinterface `NumberSink` is a functional interface; the comment denotes the single abstract method that is inherited from the superinterface `Sink<Number,Number>`.

Intersection of Functional Interfaces

An interface can be derived from several superinterfaces. If the superinterfaces are functional interfaces, the resulting intersection type is a functional interface, too, if the intersection contains a single abstract method. This abstract method may even be generic.

These cases will be rare in practice, as it is only an issue if the functional superinterfaces have single abstract methods with the same name and matching signatures. Belowe are a couple of examples of these corner cases.

Example of the intersection two functional interfaces:

```
@FunctionalInterface
interface Printable {
    void print(String s);
}
@FunctionalInterface
interface Formatter {
    void print(String s);
}
@FunctionalInterface
interface PrettyPrinter extends Printable, Formatter {
  //  void print(String s);
}
```

Quite obviously the intersection type `PrettyPrinter` has a single abstract method since the two superinterfaces have identical abstract methods. The comment in the subinterface shows the signature of the inherited single abstract method.

Here is a more complex example that involves three functional superinterfaces, two of which are generic interfaces. The three functional interfaces are:

```
@FunctionalInterface
interface RawProducer {
    Object produce();
}
@FunctionalInterface
interface GenericProducer<S> {
    S produce();
}
@FunctionalInterface
interface Source<T> {
    T produce();
```

```
}
```

Each of the three functional interfaces has a different `produce` method (with different return type). We consider the two subinterfaces

Example of the intersection of several functional interfaces:

```
@FunctionalInterface
interface ObjectProducer extends RawProducer,
                                 GenericProducer<Object>,
                                 Source<Object> {
  //  Object produce();
}
```

In this example the three superinterfaces single abstract methods collapse into a single signature in the subinterface `ObjectProducer`. The comment in the subinterface shows the signature of the inherited single abstract method.

Example of the intersection of several functional interfaces with a generic single abstract method:

```
@FunctionalInterface
interface Producer<S> extends RawProducer,
                              GenericProducer<S>,
                              Source<S> {
  //  S produce();
}
```

In this example it is less obvious why the subinterface `Producer<S>` is a functional interface. The inherited single abstract methods do not have identical signatures, but the method signatures are compatible enough to yield a functional interface as the intersection type. This is because the signature `<S extends Object> S produce()` is equivalient to `<T extends Object> T produce()` and both are a subsignature of `Object produce()`.

As you can tell form the last example in particular, the rules regarding subsignatures are fairly complex. There are many issues involved: the relationship between generic types and raw type as well as issues of substitutable return types and compatible throws clauses. If you are interested in the details, the language specification would be the best source for further information.

For all practical purposes the compiler will figure out whether the intersection of functional interfaces is still functional. In case of doubt, use the `@FunctionalInterface` annotation: qualify the subinterface with the annotation and the compiler swill raise a compile-time error message if the subinterface is not functional.

Target Typing

Lambda expressions and method/constructor references conceptually denote functions, but Java has no such thing as function types. Instead, lambda expressions and method/constructor references must be converted to *functional interface types*. We discussed the need for functional interface types and related details in the section on "Functional Interfaces". Simply put, a functional interface type is an interface with one abstract method.

We also mentioned that the compiler infers the functional interface type, to which a lambda expression or method/constructor reference is converted, from the context in which it appears. This context dependent type inference process is called *target typing*. In the following we will discuss how target typing works. Let us start by clarifying a couple of terms.

Definition

In Java, every expression must have a type. The expression's type is determined by the compiler. The type deduction process performed by the compiler depends on the nature of the expression. Java has two sorts of expressions:

- For so-called *standalone expressions* the type deduction can be performed by just analyzing the expression. Examples of standalone expressions are `array.length`, `i+5`, or `obj.getClass()`.
- For so-called *poly expressions* the type deduction requires analysis of both the expression and the context in which the expression appears. Poly expressions in isolation, i.e., without a context, have no type. An example of a poly expression is `new HashSet<>()`. It can mean different things in different context.

When an expression appears in a context, its type must be compatible with a type expected in that context. The expected type is called the *target type*.

The expression itself has a *deduced type*. For poly expressions the deduced type can be influenced by the target type. The compiler analyses the context, determines the target type and deduces a type for the expression that is equals or convertible to the target type. In contrast, a standalone expression's type is always independent of the target type; it just has to be convertible to the target type.

Lambda expressions and method/constructor references are always poly expressions, i.e., their type is deduced by the compiler from the enclosing context in which they appear. Lambda expressions and method/constructor references are slightly different from other types of poly expressions:

their deduced type is not just convertible, but equal to the target type, and

the target type cannot be an arbitrary type, but must be a functional interface type.

Before we address the process of target typing in further detail we take a look at Java expressions in general and how the compiler determines the type of an expression. This is background information that aids the understanding of type inference in general and for lambda expressions and method/constructor references in particular. A basic understanding of type inference may be helpful in situations where the type inference process fails. Failure might occur for various reasons, e.g. due to incorrect syntax, due to limitations of the compiler's type inference logic, or because of an insufficient context. With a basic understanding of type inference you will be capable of figuring out why type inference fails and how to work around it.

If you are already familiar with poly expressions and the Java compiler's type deduction strategies you might want to skip the subsequent sections and continue with the section on "Target Typing for Lambda Expression".

Classification of Expressions

For each expression in the source code the compiler must determine the type of the expression. It applies different strategies for different kinds of expressions. In Java there are two types of expressions

- *Standalone expressions*. These are expressions whose type can be determined entirely from the contents of the expression.
- *Poly expressions*. These are expressions that can have different types in different contexts. A poly expression's type is determined by the compiler from the context in which the poly expression is defined.

Determining an expression's type is comparatively easy for constant and standalone expressions and much more challenging for poly expressions. Let us take a look at examples for the various types of expressions.

Standalone Expressions

Most expressions are standalone expressions. Here are a couple of examples:

```
List<String> list = new ArrayList<String>();                    //1
list.add("Emma");                                               //2
Object ref = list.get(0);                                       //3
```

The expression `new ArrayList<String>()` in line `//1` is a standalone expression. Even in isolation the compiler can tell that the expression `new ArrayList<String>()` has the type `ArrayList<String>`. The expression's type is always the same, regardless of the context in which it appears. It does not matter whether it appears as the right-hand side of an assignment - like in the example above - or as the argument of a method invocation or in a cast expression. The type is always the same.

The expression `"Emma"` in line `//2` has a constant value and the compiler can immediately tell that its type is `String`.

The expression `list.get(0)` in line `//3` is a standalone expression, too. The compiler knows that the type of the list variable is `List<String>`, that the `get` method's return type is `String`, and concludes that the type of the entire expression is `String`. This, too, is independent of the context.

Poly Expressions

Java has a category of types called *poly expressions*. The type of a poly expression varies depending on the context in which the expression appears. The type is not specified by the programmer (in terms of Java syntax), but instead inferred by the compiler. Lambda expressions and method/constructor references are an example of poly expressions, but they are not the only poly expressions in Java.

Here is an overview of all poly expressions in Java along with the context in which they are permitted to appear. Subsequently we will discuss type inference for each of the poly expressions.

Poly Expression	Example	Context
Instance creation expression using a "diamond operator"	`new List<>`	assignment or method invocation
Invocation of a generic method or	`Collections.emptySet()`	assignment or method invocation

constructor		
Conditional operator expression	`isSequential ? new HashSet<>() : Collections.synchronizedSet (new HashSet<>())`	assignment or method invocation, unless both operands produce primitives (or boxed primitives)
Method or constructor references	`String::compareToIgnoreCase`	assignment, method invocation, or cast
Lambda expressions	`(i,j) -> i<j`	assignment, method invocation, or cast

Poly Contexts

As explained above a poly expression need a context from which the compiler can infer the poly expression's type. We call such a context a *poly context*. Before we take a closer look at the various poly expressions and how their target type is deduced, let us see in which contexts are poly contexts and which ones are not.

- In an *assigment context* the poly expression appears on the right-hand side of the assignment operator `'='`. The target type is the type of the left-hand side of the assignment.

  Example `TargetType variable = poly_expression;`

- In a *method invocation context* the poly expression appears as an argument in a method or constructor call. The target type is the declared type of the corresponding method parameter.

  Example: `ReturnType methodName(TargetType arg);`
  `methodName(poly_expression);`

- In a *cast context* the poly expression is preceeded by a cast operation, i.e., a type enclosed in parentheses. The cast's target type serves as the poly expression's target type.

  Example `(TargetType) poly_expression`

- A *return context*, i.e., when a poly expression appears as the expression after the `return` keyword in a method body, is considered an assignment context. The target type is the method's declared return type.

  Example: `TargetType methodName() { return poly_expression; }`

- A *receiver context* in a method invocation or field access, i.e., when a poly expression appears before the member selection symbol `'.'`, is not considered a context for a poly expression. Allowing this context would add another dimension to the complexity of the type inference algorithm, since the target type cannot be easily derived.

  Examples:
  ```
  poly_expression.field
  poly_expression.method()
  ```

- A *loop context,* i.e., when a poly expression appears as the expression in an enhanced for loop, is *not* considered a context for a poly expression. This is as if the expression were a receiver, namely `exp.iterator()` (or, in the array case, `exp[i]`).

  Example:
  ```
  for (Type variable : poly_expression) { … }
  ```

- A *string context* is *not* considered a context for a poly expression. It does not provide any useful information for type inference, because every value can be converted to a `String`.

  Example:
  ```
  "prefix." + poly_expression
  ```

- *Numeric and boolean contexts* (e.g. loop conditions, assert operands, binary expression operands) are *not* considered contexts for a poly expression. The reason is that some poly expressions cannot target a primitive type (instance creation expressions, lambda expressions, method references) and working out proper type inference for the remaining poly expressions would be quite complex without much payoff.

  Example:
  ```
  if (poly_expression) …
  5L + poly_expression
  ```

In a context that the compiler considers not a type inference context for a given poly expression is simply ignored. The compiler simply does not take any information from the context for deduction of missing type information.

Target Typing for Poly Expressions

In this section we take a look at type inference for the various types of poly expressions.

Target Typing for Instance Creation Expressions with the "Diamond Operator"

Instance creation expressions are poly expressions when they use the "diamond operator". Let us take a look at an example. It shows the

identical expression is two different contexts. In each context the expression has a different deduced type.

Example of an instance creation with "diamond operator":

```
Collection<Object> objs = new ArrayList<>();            //1
List<String>       list = new ArrayList<>();            //2
```

The expression `new ArrayList<>()` in line `//1` and `//2` is a poly expression. It is a `new` expression (aka instance creation expression) for a generic type that uses a diamond `"<>"` in lieu of type arguments. In other words, the type parameter for the generic type `ArrayList` has been omitted. The diamond operator turns the expression into a poly expression. The compiler must figure out the missing type parameter before the type of the entire instance creation expression can be determined.

In the example above the exact same new expression appears in two different contexts. In each context it has a different inferred type. In both cases the context is an assignment context, i.e., the poly expression appears on the right-hand side of an assignment. For type inference the compiler takes a look at the left-hand side of the assignment and finds the target type, i.e., the type that is required in the given assignment context. It then checks whether a type parameter can be found for the right-hand side that yields a compatible type. This way the compiler deduces that the poly expression must be of type `ArrayList<Object>` in line `//1` and of type `ArrayList<String>` in line `//2`.

Instance creation expressions with "diamond operator" may appear in an assignment context, like in the example above, or in a method invocation context, i.e., as the argument of a method. Other contexts are not considered for type inference. If, for instance, the instance creation expression appears in a casting context the compiler simply ignores the context information and performs the type inference as though there were no context at all. Here is an example:

Example of an instance creation expression with "diamond operator" in a casting context:

```
List<Long> list = (List<Long>) new ArrayList<>();        // error
```

The compiler ignores the cast context because casting is not a valid context for type inference of an instance creation expression. It deduces `ArrayList<Object>` as the type of the new expression (as though there were no context at all) and then complains about incompatible types because an `ArrayList<Object>` cannot be converted to a `ArrayList<Long>`.

The error message can be avoided by not using the diamond operator, but providing the correct type parameter.

The same example, but this time corrected:

```
List<Long> list = new ArrayList<Long>();                 // fine
```

The example compiles because the instance creation expression `new ArrayList<Long>()` is no poly expression and for this reason no context dependent type inference is needed.

Target Typing for Invocation of Generic Methods

The invocation of a generic method can be a poly expression. Below is an example that uses the identical method invocation in two different contexts. In each context the expression has a different deduced type.

Example of an invocation of a generic method:

```
Collection<Object> objs = new ArrayList<>();
List<String> list = new ArrayList<String>();
objs      = Arrays.asList("Emma", "Otto", "Lilo");      //1
list.addAll(Arrays.asList("Emma", "Otto", "Lilo"));     //2
```

The expression `Arrays.asList("Emma", "Otto", "Lilo")` in line `//1` and `//2` is a poly expression. It is the invocation of the generic `asList` method of class `Arrays`. Generic methods are usually invoked without specifying the method's type parameter(s). In this case the compiler must infer the missing parameter(s).

In line `//2` the poly expression appears in an assignment context, namely assignment to a left-hand side of type `Collection<Object>`. In line `//2` it appears in an invocation context, i.e., as the argument of the `addAll` method of class `List<String>`.

In the assignment context the compiler applies the same strategy as described above. It takes a look at the left-hand side of the assignment and figures out that the target type is `Collection<Object>`. The right-hand side will produce the compatible type `List<Object>` when the missing type parameter is inferred as type `Object`. For this reason the deduced type for the method invocation expression `Arrays.asList("Emma", "Otto", "Lilo")` is `List<Object>`.

In the method invocation context in line `//2` the compiler first determines the target type by figuring the declared argument of the invoked `addAll` method. It finds that the declared argument type of the `addAll` method in class `List<String>` is `Collection<String>`. It means that the result of the `asList` method must be compatible to the target type `Collection<String>`. This can be achieved if the type parameter of the generic `asList` method is `String`. Eventually, the type of the entire poly expression `Arrays.asList("Emma", "Otto", "Lilo")` is deduced as `List<String>`.

Again, the compiler deduces different types for the exact same method invocation expression depending on the context: the poly expression must be of type `List<Object>` in line `//1` and of type `List<String>` in line `//2`. Note that

in both examples the target type (`Collection<Object>` and `Collection<String>`) differs from the deduced type (`List<Object>` and `List<String>`). The deduced type is compatible to the respective target type and the compiler automatically applies the necessary conversions.

Invocations of a generic method may appear in an assignment context or in a method invocation context, as illustrated in the example. Other contexts are not considered for type inference.

Target Typing for Conditional Operator Expressions

The ternary conditional operator `"?:"` can be a poly expression. Below is an example that uses the identical conditional expression in two different contexts. In each context the expression has a different deduced type.

Example of conditional operator expression:

```
Set<String> stringSet
  = isSequential
  ? new HashSet<>()
  : Collections.synchronizedSet(new HashSet<>());            // 1
Set<Number> numberSet
  = isSequential
  ? new HashSet<>()
  : Collections.synchronizedSet(new HashSet<>());            // 2
```

The conditional operator expression in line `//1` and `//2` is a poly expression because both its operands are poly expressions. Before Java 8, such an expression was illegal; the conditional operator was not considered as a context for type inference. Since Java 8, it is permitted.

The compiler first determines the type of the left-hand side of the assignment; the required target type is `Set<String>` in line `//1` and `Set<Number>` in line `//2`. Then the compiler pushes this type information onto the two operands of the conditional expression. Thus the required target type for both operands must be compatible to `Set<String>` and `Set<Number>` respectively.

For the first operand the compiler figures out that a type argument of `String` or `Number` respectively would yield the compatible types `HashSet<String>` and `HashSet<Number>`.

The second operand is more challenging. It is the invocation of the generic `synchronizedSet` method of class `Collections`. The compiler infers that the generic method needs a type parameter of `String` or `Number` in order to yield the compatible return types `Set<String>` or `Set<Number>`. This determines the required argument type of the generic `synchronizedSet` method and leads to the requirement that the instance creation expression for the `HashSet` must be compatible to `Set<String>` or `Set<Number>`. This can be achieved by inferring `String` and `Number` as the type parameters for the `HashSet` creation.

Eventually, the compiler deduces that the poly expression is of type `Set<String>` in line `//1` and of type `Set<Number>` in line `//2`.

Conditional operators may appear in an assignment context, as illustrated in the example, or in a method invocation context. Other contexts are not considered for type inference.

Target Typing for Method and Constructor References

Method and constructor references are poly expressions. Below is an example that uses the identical method reference in several different contexts. In each context the expression has a different deduced type.

Example of a method reference:

```
List<String> list = new ArrayList<String>();
Function<String[],String[]> mapper = list::toArray;          // 1
ThreadLocal<Object[]> names =
ThreadLocal.withInitial(list::toArray);                      // 2
Object task = (Supplier<?>)list::toArray;                    // 3
Object[] tasks = new Object[]
{(Callable<Object[]>)list::toArray};                         // 4
```

The method reference `list::toArray` is a poly expression. Method and constructor references are always poly expressions; their target type must be inferred by the compiler from the enclosing context. Hence it is a compile-time error if a method or constructor reference occurs in someplace other than an assignment context, an invocation context, or a casting context.

In line `//1` the context is an assignment context. First, the compiler checks the left-hand side and figures out what the required target type is and whether it is a functional type. Method and constructor references as well as lambda expression may only appear in a context where the target type is a functional interface type. In line `//1` the left-hand side type is `Function<String[],String[]>`, which is a functional interface type from package `java.util.function`.

Next the compiler figures out the so-called *function descriptor* of both the left- and the right-hand side and checks whether they are compatible. The function descriptor is basically the description of a method without its name and body. It consists of type parameters, formal parameter types, return types, and thrown types. The function descriptor is similar to the *function signature*. The difference is that the return type is irrelvant for the signature, but part of the descriptor.

In line `//1` the left-hand side descriptor is the descriptor of the `apply` method of interface `Function<String[],String[]>`. Its descriptor is `(String[])->String[]`, which means: it is a function that takes one argument of type `String[]` and returns a value of type `String[]` and has no type parameters (i.e., it is not generic) and does not throw any checked exceptions.

The right-hand side descriptor is the descriptor of the method reference `list::toArray`. Since the `list` variable is of type `List<String>` the compiler finds that there are two candidate methods:

- a `toArray` method without arguments that returns an `Object[]` and has the descriptor `()->Object[]`, and
- a generic `toArray` method with a `T[]` argument that returns a `T[]` with the descriptor `<T>(T[])->T[]`.

Only the second method yields a compatible descriptor, namely `(String[])->String[]` when the type parameter `T` is replaced by the type `String`. The resulting function descriptor exactly matches the left-hand side's descriptor. The functional interface on the left-hand side is then determined as the target type of the method reference on the right-hand side of the assignment.

Eventually, the poly expression `list::toArray` in line `//1` has the deduced type `Function<String[],String[]>`.

In line `//2` the context is a method invocation context because the method reference `list::toArray` is passed as an argument to the `withInitial` method of class `ThreadLocal<Object[]>`. from package `java.lang`. In this context, the compiler figures out the declared argument type of the invoked method and checks whether it is a functional interface type. The argument type of the `withInitial` method is `Supplier<Object[]>`, which is a functional interface type from package `java.util.function`. Hence the target type is `Supplier<Object[]>`.

Then the compiler again figures out the function descriptors and checks whether they match. The target type is `Supplier<Object[]>`; it has a `get` method that takes nothing and returns an `Object[]`. Its descriptor is `()->Object[]`. The method reference `list::toArray` again boils down to the two candidate methods already described above. This time the first candidate has a matching function descriptor.

Eventually, the poly expression `list::toArray` in line `//2` has the deduced type `Supplier<Object[]>`.

In line `//3` the context is a casting context. Since the assignment of a method reference to a variable of type `Object` is a context in which the compiler cannot infer a target type for the method reference from the left-hand type, we use a cast to aid type inference. For this reason the method reference `list::toArray` is preceeded by a cast with target type `Supplier<?>`. In this context, the compiler checks whether the cast's target type is a functional interface type. We have already seen that `Supplier<?>` is a parameterization of the generic functional interface type `Supplier<T>` from package `java.util.function`.

Then the compiler again compares the function descriptors. The cast's target type `Supplier<?>` has the descriptor `()->?`, which mean its `get` method takes nothing and returns an arbitrary unknown type. For the method reference `list::toArray` we again have two candidate methods, as described above. The first candidate has the function descriptor `()->Object[]`, which is compatible to the required descriptor of `()->?`.

Eventually, the poly expression `list::toArray` in line `//3` has the deduced type `Supplier<?>`.

In line `//4` the poly expression appears as an array initializer. Array initializer contexts are like assignments, except that the "left-hand side variable" is an array component and its type is derived from the array's type.

Note that in all examples the required target type and the constructor/method reference's deduced type are identical. This is not by chance; it is intended. Different from other poly expressions the type of a constructor/method reference is not just convertible to, but infact identical to its target type. The important prerequisite is that the constructor/method reference is compatible with its target type; otherwise. Compatibility depends on the function descriptor; to derive this descriptor, a type target must be a functional interface type.

Target Typing & Checked Exceptions

So far we have only considered target typing for references to methods and constructors that do not throw any checked exceptions. How does target typing work if checked exceptions are involved and the referenced methods and constructors have throws clauses? It turns out that the compiler checks throws clauses for compatibility and reports errors if they are incompatible. Let us take a look at an example.

Example of a method reference that throws checked exceptions:

```
Function<Future<Number>,Number> f1
  = Future<Number>::get;                                    //1
                          // error: incompatible throws clause

ThrowingFunction<Future<Number>,Number,Exception> f2
  = Future<Number>::get;                                    //2
```

Let us first see what the involved types and methods look like.

The method reference in our example is the `get` method of the `Future` interface defined in package `java.util.concurrent`. Here is the relevant excerpt from interface `Future`:

```
public interface Future<V> {
    V get()
        throws InterruptedException,
               ExecutionException;
```

```
    V get(long timeout, TimeUnit unit)
        throws InterruptedException,
               ExecutionException,
               TimeoutException;
}
```

The method reference appears in two assignment contexts. The left-hand side of the assignment in line `//1` uses the parameterization `Function<Future<Number>,Number>` of the functional interface `Function` defined in package `java.util.function`. Here is the relevant excerpt from interface `Function`:

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

In line `//2` the left-hand side is of a different functional interface type, namely a parameterizaton of the type `ThrowingFunction`, which is a functional interface type that looks like this:

```
@FunctionalInterface
public interface ThrowingFunction<A,R,E extends Exception> {
    R get(A arg)  throws E;
}
```

Now, let us see which role the `get` method's throws clause plays in the target typing process.

In line `//1` the target type on the left-hand side has the function descriptor `(Future<Number>)->Number`, which means it is a function that takes a `Future<Number>`, returns a `Number` as a result, and does not throw any checked exceptions.

The `Future` interface's `get` method is overloaded. The two candidates have the descriptors

```
(Future<Number>,TimeUnit)->Number throws TimeoutException,
                                         ExecutionException,
                                         InterruptedException
```

and

```
(Future<Number>)->Number throws ExecutionException,
                                InterruptedException
```

The first candidate does not match at all because the number of required arguments is different (two arguments vs. one argument). The second candidate has an almost compatible descriptor; the argument list and the return type match, only the `throws` clause is different from the left-hand side's descriptor. The left-hand side requires that the function must not throw checked exceptions while the method reference on the right-hand side does raise checked exceptions. Due to the incompatible `throws` clause the compiler issues an error message.

In line `//2` the left-hand side has the descriptor `(Future<Number>)->Number throws Exception`. Its clause `throws Exception` is compatible to the method reference's clause `throws ExecutionException, InterruptedException` because `Exception` is a common supertype of both `ExecutionException` and `InterruptedException`.

The compiler deduces that the poly expression `Future<Number>::get` in line `//2` has the target type `ThrowingFunction<Future<Number>,Number, Exception>`.

Exception Tunnelling - Wrapping Checked Exceptions into Runtime Exceptions

Since most functional interface types defined by the JDK and in particular all functional interface types in package `java.util.function` do not permit checked exceptions it is common practice to wrap any checked exceptions raised by a lambda expression or method/constructor reference into an unchecked `RuntimeException`.[6]

Here is an example of such a wrapper. We define a runtime exception type `Unchecked` and an adapter operation that turns a function with checked exceptions into a function that only throws a runtime exception.

Example of wrapper for checked exceptions:

```
@FunctionalInterface
public interface ThrowingFunction<A,R,E extends Exception> {
    R get(A arg)  throws E;

    static <A,R,E extends Exception> Function<A,R>
      makeNonThrowing(ThrowingFunction<A,R,E> f) {
        return (A arg) ->  {
           try { return f.get(arg); }
           catch (Exception e) {throw new RuntimeException (e);}
        };
    }
```

Using the adapter operation `makeNonThrowing()` we can turn the exception throwing method reference `Future<Number>::get` into a non-throwing function that is compatible to the functional interface type `Function` from package `java.util.function`:

```
Function<Future<Number>,Number> f3
  = ThrowingFunction.makeNonThrowing(Future<Number>::get);     //3
```

In line `//3` the method reference `Future<Number>::get` appears in a method invocation context. The invoked method `makeNonThrowing` is a generic method.

---

6 Another example of exception tunnelling can be found in the section on "Checked Exceptions".

The first step is that the compiler performs type inference for the `makeNonThrowing` method and figures out what its type parameters `A`, `R`, and `E` must be. The compiler infers the target type `Function<Future<Number>, Number>` from the left-hand side of the assignment and concludes that the `makeNonThrowing` method's type parameters must be `A:= Future<Number>` and `R:= Number`. Since there are no particular requirements for the type parameter `E`, i.e. the exception type, the compiler provisionally uses the upper bound and infers `E:= Exception`.

The second step is type inference for the method reference `Future<Number>::get`. The target type is the `makeNonThrowing` method's declared argument type. Given the already inferred type parameters the `makeNonThrowing` method has the declared argument type `ThrowingFunction<Future<Number>,Number,Exception>` with the descriptor `(Future<Number>)->Number throws Exception`. We have already seen above that this function descriptor is compatible to the function descriptor of `Future<Number>::get`.

The compiler deduces that the poly expression `Future<Number>::get` in line `//3` has the deduced type `ThrowingFunction<Future<Number>,Number, Exception>`.

Target Typing & the Return Type

During target typing the compiler does not only check for compatible `throws` clauses, but also checks for compatibility of the return types and reports errors if they are incompatible. Let us take a look at an example.

Example of a method reference with a return type:

```
ThrowingFunction<Future<Integer>,Number,Exception> f4
  = Future<Integer>::get;                                    //4
```

We again use a to the `get` method of the `Future` interface defined in package `java.util.concurrent`.

The target type on the left-hand side of the assignment has the descriptor `(Future<Integer>)->Number throws Exception`. The method reference on the right-hand side has the descriptor `(Future<Integer>)->Integer throws ExecutionException, InterruptedException`. In addition to the different `throws` clauses also the return type differs: the left-hand side requires a return type of `Number` whereas the method reference has the return type `Integer`. Since `Integer` is a sub-type of `Number` the functions descriptors are compatible.

The compiler deduces that the poly expression `Future<Number>::get` in line `//4` has the deduced type `ThrowingFunction<Future<Integer>,Number, Exception>`.

Target Typing for Lambda Expressions

Lambda expressions are always poly expressions, i.e., their type depends on the context in which they are declared and the target type is always inferred by the compiler.

In the following we will discuss target typing for lambda expressions in various contexts.

Assignment Context

Example of a lambda expression in an assignment context:

```
BiPredicate<String,String>         sp1
  = (s,t) -> s.equalsIgnoreCase(t);                    // 1
BiFunction<String,String,Boolean> sp2
   = (s,t) -> s.equalsIgnoreCase(t);                   // 2
```

In line `//1` and `//2` we see the lambda expression in two different assignment contexts. In analogy to method references, the compiler checks the left-hand side and figures out what the required target type is and whether it is a functional interface type.

In line `//1` the left-hand side type is `BiPredicate<String,String>`, which is a parameterization of a generic functional interface type from package `java.util.function`. Next the compiler figures out the function descriptor of both the left- and the right-hand side and checks whether they are compatible. The right-hand side descriptor is the descriptor of the `test` method in class `BiPredicate<String,String>`. Its descriptor is `(String,String)->boolean`.

The lambda expression's descriptor is `<X,Y>(X,Y)->boolean`, which means it takes two arguments of yet unknown type and returns a `boolean` value. When the unknown types are inferred as type `X:=String` and `Y:=String` then the lambda expression's descriptor matches the left-hand side descriptor.

Hence, the poly expression `(s,t) -> s.equalsIgnoreCase(t)` in line `//1` has the deduced type `BiPredicate<String,String>`.

In line `//2` the left-hand side type is `BiFunction<String,String,Boolean>`, which is a functional interface type from package `java.util.function`. Its descriptor is the descriptor of the `apply` method in class `BiFunction<String,String,Boolean>`, namely `(String,String)-> Boolean`.

The lambda expression's descriptor still is `<X,Y>(X,Y)->boolean`. When the unknown types are inferred as `X:=String` and `Y:=String` and the return type is boxed to `Boolean` then the lambda expression's descriptor matches the left-hand side descriptor.

Hence, the poly expression `(s,t) -> s.equalsIgnoreCase(t)` in line `//2` has the deduced type `BiFunction<String,String,Boolean>`.

Return Context

Example of the same lambda expression in return context:

```
BiPredicate<String,String> makePredicate() {
  return (s,t) -> s.equalsIgnoreCase(t);                  // 3
}
```

In line `//3` we see the lambda expression in a return statement. This is a context similar to an assignment context. The required target type is the `makePredicate` method's declared return type. The required target type therefore is `BiPredicate<String,String>`. We have already seen above that the lambda expression is convertible to this target type.

Hence, the poly expression `(s,t) -> s.equalsIgnoreCase(t)` in line `//3` has the deduced type `BiPredicate<String, String>`.

Method Invocation Context

Example of the same lambda expression in a method invocation context

```
Predicate<String> matches(String arg)  {
  return bind1st((s,t) -> s.equalsIgnoreCase(t),arg);   // 4
}
```

In line `//4` we see the lambda expression in a method invocation context. The required target type is the invoked `bind1st` method's declared argument type. The `bind1st` method looks like this:

```
<T> Predicate<T> bind1st(BiPredicate<T,T> predicate, T first) {
  return  s -> predicate.test(first,s);
}
```

Since the `bind1st` method is a generic method, its type parameter must be inferred before its declared argument types are known. Hence the first step is type inference for the generic `bind1st` method. It appears in a return context. The target type is the `matches` method return type, which is `Predicate<String>`. From this requirement the compiler deduces that the `bind1st` method's type parameter `T` must be inferred as `T:=String`.

Then the compiler pushes this requirement onto the `bind1st` method's arguments, which then must be of type `BiPredicate<String,String>` and `String`.

This way, the compiler figures out that target type for the lambda expression is `BiPredicate<String,String>`. We have already seen above that the lambda expression is convertible to this target type.

Hence, the poly expression `(s,t) -> s.equalsIgnoreCase(t)` in line `//3` has the deduced type `BiPredicate<String, String>`.

Casting Context

Example of the same lambda expression in context that demands casting:

```
BiFunction<String,String,Integer> sp3
   = ((s,t) -> s.equalsIgnoreCase(t)).andThen(b->b?1:0);     // 5
                                      // error: illegal context

BiFunction<String,String,Integer> sp4
   = ((BiFunction<String,String,Boolean>)
      (s,t) -> s.equalsIgnoreCase(t))                        // 6
      .andThen(b->b?1:0);
```

In line `//5` we see the lambda expression in an illegal context. A lambda expression is not allowed as the receiver in a method invocation, field access, etc., i.e., it must not appear on the left-hand side of the member selection symbol `'.'`. In fact, no poly expression is permitted in this context. The reason is that it would add another dimension to the complexity of the compiler's type inference algorithm.

In such a situation casting comes to the rescue, as demonstrated in line `//6`. The target type for deduction of the method reference's type is the cast's target type `BiFunction<String,String,Boolean>`. We have already seen above that the lambda expression is compatible to the target type of the cast.

Hence, the lambda expression in line `//6` has the deduced type `BiFunction<String,String,Boolean>`.

Casting to an Intersection Type

A special case of a casting context is a situation where the target type of the cast is a so-called *intersection type*. Intersections types are a new feature since Java 8; in earlier versions of Java they were illegal. Casts to an intersection type are necessary when a lambda expression (or method/constructor reference) must be assigned to a variable whose type is an empty marker interface type, like for instance `java.io.Serializable`.

Let us consider an example. It uses the same lambda expression as before, but this time in a casting context with an intersection type.

Example of a lambda expression in a casting context with an intersection type:

```
Serializable f1
  = (s,t) -> s.equalsIgnoreCase(t);                          // 7
                  // error:  Serializable is not a function type
Serializable f2
  = (BiPredicate<String,String> & Serializable)              // 8
    (s,t) -> s.equalsIgnoreCase(t);
```

In line `//7` we see the lambda expression in an assignment context where the target type is `Serializable`. The compiler rightly complains because the empty

marker interface `Serializable` is not a functional interface type. So, what can we do to force the lambda expression to conform to the `Serializable` interface? We could define a `SerializableBiPredicate` helper interface, which would be a subinterface of both the `Serializable` and the `BiPredicate` interface and could serve as the target type. Fortunately, this is not necessary. Since Java 8, we can cast to so-called *intersection types*.

An intersection type is a list of types like `BiPredicate<String,String> & Serializable` in line `//8`. An intersection type can appear as the target type of a cast expression. The resulting target type is a synthetic type that is a subtype of all specified types. In particular, if the list includes an empty marker interface, the synthetic intersection type is also a subtype of the marker interface.

Intersection types did already exist before Java 8. They were internally used by the compiler in the processes of wildcard capture and type inference. But before Java 8 it was not possible to express an intersection type directly in a Java program, as no syntax supported this. Since Java 8, intersection types can be used as target types of casts.

They are particularly useful as the casting context for a lambda expression or a method/constructor reference. As explained in the example above, it provides a simple way of making a lambda expression or a method/constructor reference conform to an interface such as `Serializable`. If an intersection type is used in a casting context for a lambda expression or a method/constructor reference, then the intersection must be a functional interface type. It typically means that one type is a functional interface and the others are marker interfaces.

Eventually, the result of the target typing process is that the lambda expression in line `//8` has the deduced type `BiPredicate <String,String> & Serializable`.

Wrap-Up

In this section we looked at the type inference process that the compiler performs for poly expressions in general and lambda expressions and method/constructor references in particular. The selected examples were fairly simple. In reality the process is more complex, especially when several type deduction processes must be performed simultaneously for a given expression. Mind, the compiler applies several type deductions:

- *overload resolution* (i.e. selecting a matching method from a set of candidate methods with the same name and different signatures) ,
- *type argument inference* (i.e. figuring out which type parameters must be used for parameterization of a generic type or method if the type

parameters are omitted in a generic method invocation or instance creation*)*, and

- *target typing* (i.e. deducing a lambda expression's or method/constructor reference's matching functional interface type).

The likelihood for type inference failures increases the more deductions must be performed for a given expression. In the next section we want to explore some of these problematic situations where type inference fails.

Type Inference Issues

Occasionally, type inference fails and the compiler may abort the type deduction process with an error messages. In these situations you need to understand the issue and must find a workaround.

Coping with type inference issues has two aspects: usage and design of an API.

- *Usage*. As the user of an API that is prone to type inference failures you need to figure out a workaround. Often the problem can be solved by adding casts in the right places or replacing implicit with explicit lambdas (an implicit lambda being a lambda expression without specification of the arguments types).
- *Design*. As the designer of an API you might want to set up the API in an manner that avoids type inference failures in the first place. This can be achieved by avoiding overloading and/or avoiding wildcards or generics in general.

In this section we want take a look at a couple of situations where type inference fails. We start with issues that occur frequently in practice and proceed to more esoteric situations that are rare in practice.

Common Type Inference Issues

In the section on "Poly Expressions" we learnt that in isolation, i.e. without a context, poly expressions such as lambda expressions and method/constructor references do not have a type. For instance, the lambda expressions `s -> s.length()` is meaningless unless it appears in a context from which the compiler can deduce the type of `s`.

In the section on "Poly Contexts" we have seen that there are several contexts permitted as type inference context for a lambda expression or method/constructor reference: assignment, method invocation, and cast. The type inference process is comparatively easy in an assignment or cast

context, but can be fairly complex in a method invocation context, especially when the invoked method is overloaded.

In the following we will first look into harmless poly contexts and then into the problematic ones. The harmless situation are of interest because the problematic ones can be resolved by turning them into harmless ones.

Harmless Poly Contexts

Let us study an example. Consider the functional interfaces `Function`, `ToIntFunction`, `ToLongFunction`, and `ToDoubleFunction` from package `java.util.function`:

```
interface         Function<T,R> { R      apply        (T arg); }
interface    ToIntFunction<T>   { int    applyAsInt(   T arg); }
interface   ToLongFunction<T>   { long   applyAsLong  (T arg); }
interface ToDoubleFunction<T>   { double applyAsDouble(T arg); }
```

The lambda expressions `s -> s.length()` can be compatible to either of them. Let us say, the `length` method in the lambda expression denotes the `length` method declared in the `CharSequence` interface and defined in any of its subtypes (`String`, `StringBuiler`, `StringBuffer`, etc.). The `length` method returns an `int` value.

The deduced type of the lambda expression can be any of the functional interfaces above, provided that the type parameter `T` is replaced with `CharSequence` or a subtype thereof and the type parameter `R` with a type that can store an `int`. Examples of compatible functional types include: `ToIntFunction<String>`, `ToLongFunction<CharSequence>`, `Function <StringBuilder,Integer>`, `Function<String,Object>`, and many more. When the lambda expression appears in a poly context then the compiler must infer one of these compatible types.

In an assignment or cast context the type inference is fairly easy because the target type is clearly defined. Here are some examples.

Examples of assignment context:

```
Function<String,Object>      f1 = s -> s.length();          //1
ToLongFunction<CharSequence> f2 = s -> s.length();          //2
```

Examples of cast context:

```
Object o = (ToIntFunction<StringBuilder>)  s -> s.length(); //3
       o = (Function<CharSequence,Number>) s -> s.length(); //4
```

In the assignment context the target type is the type on the left-hand side of the assignment. In the cast context the target type for type inference is the target type of the cast. In all cases the target type is clearly defined.

In `//1` the compiler deduces `String` as the type of `s` and subsequently checks whether the return type of the lambda expression (namely `int`) is compatible to the return type in the target type's signature (namely `Object`). With autoboxing `int` is convertible to `Object` and the type inference succeeds.

In a similar fashion, the compiler deduces `String`, `StringBuilder` or `CharSequence` as the type of `s` and subsequently checks for the return type compatibility, i.e. whether `int` is compatible to `long`, `int`, or `Number` respectively.

Type inference is equally easy if the inference context is a method invocation where the method in question is not overloaded.

Example of simple method invocation context:

```
interface I { double transform(ToDoubleFunction<String> f); }

I iRef = … some implementation of iterface I …
iRef.transform(s -> s.length);                                    //5
```

The target type for type inference is the method's declared argument type `ToDoubleFunction<String>`. The compiler deduces `String` as the type of `s` and finds that the return types `int` and `double` are compatible.

Problematic Poly Contexts

The type inference process is more complicated when the invoked method in an invocation context is an overloaded method. Below is an example where the lambda expression is passed to an overloaded method.

Example of a method invocation context with overloading:

```
interface I<T> {
  <R> R  map(Function<T,R> f);
  int    map(ToIntFunction<T> f);
  long   map(ToLongFunction<T> f);
  double map(ToDoubleFunction<T> f);
}

I iRef = … some implementation of iterface I …
iRef.map(s -> s.length());  // error: ambiguous                   //6
```

The compiler considers all four `map` methods as candidates for overload resolution, which means that there are four different target types for the lambda expression. In such a situation the type inference process fails and the compiler reports an ambituity. In fact, the compiler already complains about an ambiguity if there is *more than one* viable target type; it does not even take four candidates; two overloaded methods suffice for an ambiguity.

A Note on Language Design Decisions (Related to Type Inference)

You might wonder why the compiler isn't smart enough to avoid type inference failure. The answer is: the compiler *is* smart enough; it was a deliberate design decision to keep type inference simple - at the expense of more frequent type inference failure.

More complex overload resolution schemes are conceivable that would avoid type inference failure in situations like the one above. They were discussed and even tentatively implemented during the design of lambdas. In principle, the compiler can apply all kinds of magic in order to come up with a most specific target among several viable candidates. It could for instance take the return type into account, or the throws clauses for that matter, or perform speculative strategies with corresponding backtracking. All this can be done by a compiler, but it was decided that the compiler should not do it. All experiments with more sophisticated type inference and overload resolution schemes were eventually discarded.

The language designers decided in favour of a relatively simple type inference process (at the expense of more frequent type inference failure) for two reasons: a) in order to keep type inference understandable for its users and b) for more robust code.

Ad a) The more complex the type inference process is the more difficult it is for its users to track it down. Even a more sophisticated type inference scheme will occasionally fail. Failure will be rare, but it can still happen. In case of type inference failure under a more complex scheme, the programmer hardly has a chance to figure out a solution due to the overall complexity. The designers wanted to avoid inexplicable magic and decided to keep type inference comprehensible.

Ad b) Very sophisticated type inference schemes can produce britle code, where a seemingly harmless modification of one piece of code can change the result of type inference in another piece of code. It means that a small change here can trigger invocation of another method there (in a seemingly unrelated piece of code). Such unexpected side effects are usually undesired and difficult to track down in presence of a complex type inference scheme. The designers tried to eliminate surprising side effects.

Coping With Type Inference Failure

Situations of type inference failure have two aspects: either you are the user of a method that leads to the failure, or you are the designer of such a method. Correspondingly, there are two strategies for tackling the problem:

- *User site workaround:* Avoid type inference, e.g. use explicit lambdas or method references.
- *Declaration site workaround:* Avoid overloading on functional interface types, i.e. rename the overloaded methods.

Let us explore the workarounds using the example of an overloaded `map` method and the lambda expression `s->s.length()`.

User Site Workarounds

The user of a potentially problematic API can avoid the type inference failure by supplying more type information to the compiler. The user site workaround for the example that we have been exploring earlier can look like this:

```
interface I<T> {
  <R> R  map(Function<T,R> f);
  int    map(ToIntFunction<T> f);
  long   map(ToLongFunction<T> f);
  double map(ToDoubleFunction<T> f);
}

I iRef = … some implementation of iterface I …
iRef.map((String s) -> s.length);                    // fine   //1
iRef.map(String::length);                            // fine   //2

iRef.map((Function<String,Integer>)s -> s.length());// fine   //3
iRef.map((ToIntFunction<String>)s -> s.length());   // fine   //4

ToLongFunction<String> mapper = s -> s.length();    // fine   //5
iRef.map(mapper);
```

The idea is to reduce the need for type inference in the first place. Instead of having the compiler infer the lambda's signature, we can support the compiler by supplying type information explicitly. The resolutions above all assume that the lambda expression `s -> s.length()` is supposed to work on strings and uses the `length` method of class `String`.

The first workaround uses an explicit lambda expression (see `//1`) instead of an implicit one. An explicit lambda expression has its argument types explicitly specified so that the compiler need not infer them.

The solution in `//2` does essentially the same: it uses the method reference `String::length`, which has the signature `(String)->int`. Again, the compiler need not infer the argument type because the method reference already supplies it.

The approaches in `//3` and `//4` are the least attractive ones: they turn the method invocation context into a cast context by casting the entire

lambda expression to an appropriate target type like `Function<String,Integer>`, `ToIntFunction<String>`, or any other compatible functional interface type.

The solution in `//5` introduces a variable to which the lambda expression is assigned. Thereby it turns the method invocation context into an assignment context.

The first two solutions have in common that they describe the lambda more precisely by supplying the lambda's argument type (explicitly in the lambda expression and implicitly in the method reference).

The last three solutions have in common that they turn the problematic poly context (invocation of an overloaded method) into a harmless poly context (cast or assignment).

Application of the User Site Workarounds in Practice

Let us explore an example where the user site strategy described above is needed. The `Comparator` interface from package `java.util` has five overloaded versions of a `comparing` method. When calling the overloaded method it is often best to use explicit lambdas (rather than implict ones) in order to avoid the ambiguity error messages.

Here is an example using the `comparing` method:

```
List<String> strs = Arrays.asList("i","xzy","X","FF80A0");
strs.sort(Comparator.comparing(s->s.length()));          // error
strs.sort(Comparator.comparing((String s)->s.length())); // fine
strs.sort(Comparator.comparing(String::length));         // fine
strs.sort(Comparator.comparing(
         (ToIntFunction<String>)s->s.length()));         // fine
System.out.println(strs);
```

With an implicit lambda expression the invocations of the `comparing` methods fails due to an ambiguity error message. With an explicit lambda expression or a method reference the type inference succeeds.

Step-by-Step Elimination of Compilation Errors

However, in practice, matters can get rather tedious at times. In the following we want to demonstrate a strategy for finding workarounds even in more complex situations than the one above. We will walk you step-by-step through the process of eliminating a compilation error. Along the way we will explain why the compiler complains and will try out various approaches for getting rid of the compiler messages.

Here is our case study (note that it does not yet compile):

```
List<String> strs = asList("ivn.txt", "Spam.pdf", … );
```

```
strs.sort(Comparator
  .comparing(s->{ int dot=s.lastIndexOf('.');
                  return (dot>=0)?s.substring(dot):"";
                })
  .thenComparing(s -> s.length())
  .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
);
```

Before we address the compilation error let us see what the code snippet does. A list of strings is shall be sorted with a composed comparator that sorts the strings first by their suffix, then by their length, and eventually according to their string content ignoring upper/lower case differences.

For composing the comparator two methods from the `Comparator` interface are used: the `comparing` method which has five overloaded versions and the `thenComparing` method which has six overloaded versions.

When we compile the code snippet then the compiler complains.

```
strs.sort(Comparator
  .comparing(s->{ int dot=s.lastIndexOf('.');           // error
                  return (dot>=0)?s.substring(dot):"";
                })
  .thenComparing(s -> s.length())
  .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
);
```

The compiler reports an ambiguity error for method `comparing`. Among the five overloaded versions of the `comparing` method the compiler finds four that are applicable, namely `comparing(ToDoubleFunction)`, `comparing(ToLongFunction)`, `comparing(ToIntFunction)`, and `comparing(Function)`. The fifth one is `comparing(Function,Comparator)`; it is ruled out because it takes two arguments and we clearly provided only one argument. Note that the compiler does not take the return type into account for overload resolution purposes in order to rule out the ToDouble, ToLong, and ToInt versions. So far the compiler does not even know that the lambda takes a `String`; how can it possibly know that it returns `String`? Anyway, there is more than one applicable candidate method and the compilation fails with an ambiguity message.

Let us fix it by passing an explicit lambda expression to the `comparing` method:

```
strs.sort(Comparator
  .comparing((String s)->{ int dot=s.lastIndexOf('.');   // error
                           return (dot>=0)?s.substring(dot):"";
                         })
  .thenComparing(s -> s.length())
  .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
```

```
);
```

It still does not compile. This time the compiler knows that the lambda takes a `String` and returns a `String` and picks `comparing(Function)` as the only applicable candidate among the five overloaded versions. This overloaded version is a static generic method with two type variables. Here is its declaration:

```
static <T,U extends Comparable<? super U>>
Comparator<T>
comparing(Function<? super T, ? extends U> keyExtractor)
```

For generic methods the compiler must infer the type parameters, i.e. what `T` and `U` are supposed to be. It does so in two steps: by first taking a look at the argument provided to the generic method and then taking the context into account in which the generic method appears. The argument provided to the generic `comparing` method is the `keyExtractor`, i.e. our lambda expression, whose type the compiler does not yet fully know. The context in which the generic `comparing` method appears does not help either. The generic method appears as the receiver of the subsequent call of method `thenComparing`, i.e. it appears on the left-hand side of the method selection symbol `'.'`. This is not a valid type inference context and does not provide any information. Ultimately, the compiler fails to infer the two type parameters of the generic comparing method.

So, our next attempt could be to provide the missing type parameters (in which case we need not specify the lambdas argument type any more):

```
strs.sort(Comparator
  .<String,String>comparing(                                  // fine
                      s->{ int dot=s.lastIndexOf('.');
                           return (dot>=0)?s.substring(dot):"";
                         })
  .thenComparing(s -> s.length())                            // error
  .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
);
```

An alternative is a cast that eliminates the need for type inference:

```
strs.sort(Comparator
  .comparing((Function<String,String>)                        // fine
                      s->{ int dot=s.lastIndexOf('.');
                           return (dot>=0)?s.substring(dot):"";
                         })
  .thenComparing(s -> s.length())                            // error
  .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
);
```

Now the invocation of the `comparing` method compiles.

After solving the first problem, the compiler complains that the invocation of `thenComparing` is ambiguous. Among the six overloaded version the compiler considers four applicable, namely `thenComparing(ToDoubleFunction)`, `thenComparing(ToLongFunction)`, `thenComparing(ToIntFunction)`, and `thenComparing(Function)`. The fifth overloaded version is `thenComparing(Function,Comparator)`, which takes two arguments. Since we provided only one argument the fifth version is inapplicable. The sixth version is `thenComparing(Comparator)`. A `Comparator` takes two arguments, but we provided a lambda that takes only one argument. This rules out the sixth version of `thenComparing`.

In the end, there is more than one applicable candidate method and the compilation fails with an ambiguity message.

Let us fix it by passing an explicit lambda expression to the `thenComparing` method:

```
strs.sort(Comparator
  .comparing((Function<String,String>)
                      s->{ int dot=s.lastIndexOf('.');
                           return (dot>=0)?s.substring(dot):"";
                         })
  .thenComparing((String s) -> s.length())                // fine
  .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
);
```

With this additional information the compiler picks `thenComparing(ToIntFunction<String>)` as the best candidate. It is a non-generic method, no furhter type inference is needed, and the problem is solved.

The last invocation of `thenComparing` does not create an ambiguity because we supply a lambda with two arguments. The compiler picks `thenComparing(Comparator<String>)` because it is the only candidate that takes a function with two arguments.

Common Practice: Break Chains Down Into Single Steps

The strategy that we have been employing above for eliminating the compilation errors requires some insight into the compiler's type inference strategies. A more practical approach might be breaking a chain of method calls down into single steps by introducing extra variables for the arguments of each step. Basically, this strategy systematically eliminates the need for type inference almost entirely by reducing it to relatively simple assignment contexts. It works like this.

We would take the initial approach (that does not compile):

```
strs.sort(Comparator
  .comparing(s->{ int dot=s.lastIndexOf('.');           // error
                  return (dot>=0)?s.substring(dot):"";
```

```
                })
    .thenComparing(s -> s.length())
    .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
);
```

and would break it down into this:

```
Function<String,String> extractor1
  = s-> { int dot=s.lastIndexOf('.');
          return (dot>=0)?s.substring(dot):"";
        };
ToIntFunction<String>   extractor2
  = s -> s.length();
Comparator<String>      comparator
  = (s1,s2) -> s1.compareToIgnoreCase(s2);

strs.sort(Comparator
          .comparing    (extractor1)
          .thenComparing(extractor2)
          .thenComparing(comparator)
);
```

Often, breaking down a chain of operations like this is only an intermediate step for elimination of error messages. You might want to use the insights gained by breaking the chain down for subsequent insertion of casts. So, the ultimate result could look like this:

```
strs.sort(Comparator
          .comparing((Function<String,String>)
               s->{ int dot=s.lastIndexOf('.');
                    return (dot>=0)?s.substring(dot):"";
                  })
          .thenComparing((ToIntFunction<String>) s -> s.length())
          .thenComparing((Comparator<String>)
               (s1, s2) -> s1.compareToIgnoreCase(s2))
          );
```

Of course, we can omit all casts that the compiler does not need and would end up with a less cluttered code like this:

```
strs.sort(Comparator
          .comparing((Function<String,String>)
               s->{ int dot=s.lastIndexOf('.');
                    return (dot>=0)?s.substring(dot):"";
                  })
          .thenComparing((ToIntFunction<String>) s -> s.length())
          .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
          );
```

An Alternative: Using Method References Instead of Lambdas

A different approach that we have not yet considered is use of method references instead of lambda expressions. Let us see what happens if we use method references.

First, we define a helper method that replaces the lambda expression we used to extract the suffix from each string.

```
class Utils {
  public static String getSuffix(String s) {
        int dot = s.lastIndexOf('.');
        return suffix = (dot>=0)?s.substring(dot):"";
  }
}
```

We use this helper method to replace all lambdas by method references. Then our example looks like this:

```
strs.sort(Comparator
          .comparing(Utils::getSuffix)
          .thenComparing(String::length)
          .thenComparing(String::compareToIgnoreCase)      // error
          );
```

It almost compiles. The method references provide more information than the implicit lambdas, which eliminates some of the ambiguities. Only the reference to `String::compareToIgnoreCase` is considered ambiguous. The problem can be solved by a cast:

```
strs.sort(Comparator
  .comparing(Utils::getSuffix)
  .thenComparing(String::length)
  .thenComparing((Comparator<String>)String::compareToIgnoreCase)
);
```

Ultimately, the most readable and concise notation is probably a combination of the various approaches, for instance this one:

```
class Utils {
  public static String getSuffix(String s) {
        int dot = s.lastIndexOf('.');
        return suffix = (dot>=0)?s.substring(dot):"";
  }
}
strs.sort(Comparator
          .comparing(Utils::getSuffix)
          .thenComparing(String::length)
          .thenComparing((s1, s2) -> s1.compareToIgnoreCase(s2))
          );
```

It is not cluttered by any cast and probably the most readable solution of all.

Wrap-Up on User Site Workarounds for Type Inference Issues

If you run into type inference problems the resolution strategy is: provide more type information. This can mean:

- use explicit lambdas instead of implicit ones, i.e. explicitly specify the lambda's argument types;
- try out method references instead of lambda expressions; sometimes it helps, sometimes it doesn't;
- add casts that specify the lambda's or method reference's intended type;
- break down a chain of operations into single steps by introducing a separate variable for each lambda expression / method reference.

Declaration Site Workaround & API Design Considerations

In the previous section we discussed what a user of an API can do if he or she faces compilations errors due to type inference failure. Avoiding type inference errors is also a design issue that API designers must take into account.

Type inference failure due to ambiguity of an overloaded method can be avoided during API design already by refraining from overloading.

In order to illustrate the corresponding design option we re-visit the example which we earlier used to illustrate the type inference failures caused by overloaded methods.

Here is the example of an interface with overloaded map methods that causes type inference problems:

```
interface I<T> {
  <R> R  map(Function<T,R> f);
  int    map(ToIntFunction<T> f);
  long   map(ToLongFunction<T> f);
  double map(ToDoubleFunction<T> f);
}

I iRef = … some implementation of iterface I …
iRef.map(s -> s.length());                    // error: ambiguous
```

The declaration site workaround for eliminating the compiler error message could look like this:

```
interface I<T> {
  <R> R  map(Function<T,R> f);
  int    mapToInt(ToIntFunction<T> f);
  long   mapToLong(ToLongFunction<T> f);
  double mapToDouble(ToDoubleFunction<T> f);
```

```
}

I iRef = … some implementation of iterface I …
iRef.map(s -> s.length);                                    // fine
```

We simply rename the `map` methods and give each version a different name. With the `map` methods renamed there is no overloading any longer and the potential for ambiguities vanishes.

Is overloading evil?

This design approach bears the question whether overloading should be generally avoided. After all, there is hardly ever a compelling need to use the same method name repeatedly in the same API, except for constructors perhaps. Can't and shouldn't we always use a different name for each method?

The answer is: no, not every set of overloaded methods causes trouble. The following properties render a set of overloaded methods problematic in conjunction with type inference:

- the argument types are functional interface types, and
- the overloaded methods have the same number of arguments.

Or, conversely, a set of overloaded methods is substantially less likely to cause type inference failures if all methods have a different number of arguments and the argument types are not functional interface types.

The need for type inference is particularly pronounced for lambda expressions and method references. These can only be supplied as arguments to a method if the argument types are functional interface types. Overloaded methods that do not take lambdas (or method references) as arguments are mostly unproblematic.

Overload resolution is more likely to fail if several of the overloaded versions have the same number of arguments. If the number of arguments differs among the overloaded versions the compiler can easily rule out inapplicable candidates: every method that has the wrong number of arguments is eliminated from the candidate set. This way it is much easier to reduce the candidate set to a single, unambiguous method.

Application of the User Site Workarounds in Practice

The design approach suggested above (distinguishing methods by name rather than signature) can be found in JDK APIs. It is the approach that was for instance taken for the `Stream` interface in package

`java.util.stream`. The `Stream` interface has several `map` methods with different names (`map`, `mapToInt`, `mapToLong`, `mapToDouble`).

The downside of an API that does not use overloading is that the user must be aware of the fact that there are several methods with different names. In the case of the stream's `map` methods the user must know that it is inefficient to use the `map` method instead of the more specific `mapToInt`, `mapToLong`, and `mapToDouble` methods. The `mapTo`***`Primitive`*** methods avoid autoboxing, while the plain `map` method does box and unbox primitive type values.

Here is an example using the `map` methods from interface `Stream`:

```
List<String> strs = Arrays.asList("i", "xzy", "X", "FF80A0");
int r;
r=strs.stream().mapToInt(String::length).sum();                 //1
r=strs.stream().mapToLong(String::length).sum();  // error     //2
r=strs.stream().map(String::length).sum();        // error     //3
r=strs.stream().map(String::length).reduce(0,(i1,i2)->i1+i2); //4
```

The method reference `String::length` can be passed to all four `map` methods. As `String::length` returns an `int` value, the `mapToInt` operation is the most efficient one. It returns a primitive stream of type `IntStream`. The `mapToLong` method does basically the same; it converts the `int` return values of `String::length` to `long` values and returns a `LongStream`. The `map` operation also works. It boxes the `int` return values of `String::length` into `Integer`s and creates a stream of boxed values of type `Stream<Integer>`.

The different behaviour of the various `map` methods becomes visible when the next operation in the chain is applied. The subsequent `sum` operation works on an `IntStream` in `//1`; it calculates the sum as an `int` value. The `sum` operation in `//2` works on a `LongStream`; the sum therefore is a `long` value, which cannot be assigned to the `int` variable `r`, which causes a compile time error. In `//3` the `sum` operation is called on a `Stream<Integer>`; the regular, non-primitve streams do not have a `sum` operation, which causes a compile time error. In `//4` we calculate the sum via the `reduce` operation, which is available for all stream types; it calculates the sum from the boxed `Integer`s and performs a lot of boxing and unboxing along the way.

API Design Considerations

The example illustrates the downside of an API that refrains from using overloaded methods: the user must decide which method to invoke and might inadvertantly pick the least efficient one (like `map` instead of `mapToInt` in the example).

On the other hand, the downside of an API that uses overloading is occasional type inference failure due to ambiguities as discussed in the previous section on "Problematic Poly Contexts".

One question remains: when should an API use overloading and when is it better to refrain from overloading? To answer the question let us look at the JDK.

We have seen examples for both design choices in the JDK. The `Stream` interface does not overload, but has four `map` methods with four different names. The `Comparator` interface does the opposite; it has five overloaded `comparing` methods and six overloaded `thenComparing` methods.

The difference is that the `map` methods in interface `Stream` return different types of streams, whereas the `comparing` methods in interface `Comparator` all return a comparator of the same type. In this sense the `comparing` methods have more similarities with each other than the `map` methods have.

In addition, the various stream types returned from the four `map` methods really make a difference: they have different APIs. For instance, the primitive streams returned from the `mapTo`***`Primitive`*** methods have a `sum` operation, which does not exist in the regular `Stream<T>` returned from the plain `map` method. For this reason it is sensible that the programmer must make a deliberate decision regarding the `map` method rather than leaving the decision to the compiler's overload resolution strategies.

In contrast, a deliberate decision regarding the `comparing` method is not strictly necessary. All `comparing` methods return the same type of `Comparator` and it does not make much of a difference which one is called. Under these circumstances (plus considering the even higher number of related methods) is deems sensible to use overloading instead of different names.

Wrap-Up on Declaration Site Workarounds for Type Inference Issues

To avoid type inference failure is to some extent an API design issues. An API that does not overload methods whose argument types are functional interface types reduces the probability that its users will be confronted with type inference failures. In other words, overloading should be used carefully for method with functional argument types.

Infrequent Type Inference Issues

The following sections cover type inference issues that occur rarely in practice. Feel free to skip these sections until you come across any of these infrequent situations.

References to Overloaded Methods/Constructors

Method and constructor references refer to a name, not to a particular signature (see e.g. the section on "Reference to Constructor"). If the referenced method or constructor is overloaded the compiler must pick from the set of overloaded methods/constructors the one that matches the requirements of the poly context in which the method/constructor reference appears. Naturally, the compiler can come to the conclusion that there are several matching versions for a given context and may reject the method/constructor reference as ambiguous. It is not a common situation, but it may happen.

Here is an example of a class with two overloaded constructors:

```
class C {
  public C(CharSequence arg) { … }
  public C(Serializable arg) { … }
}
```

When we create a reference `C::new` to the overloaded constructor and the constructor references appears in a context where both overloaded versions would match, the compiler complains. Below is an example of a context that leads to a compile-time error.

Example of an overloaded constructor reference in an assignment context:

```
Function<? super String,C> f;
f = C::new;                                          // error
f = (Function<CharSequence,C>)C::new;                // fine
f = (Function<Serializable,C>)C::new;                // fine
```

The compiler finds the constructor reference in an assignment context. The target type is a wildcard parameterization of the functional interface `Function` from package `java.util.function`. The two constructors have the signatures `(CharSequence)->C` and `(Serializable)->C`. They are compatible with the parameterizations `Function<CharSequencer,C>` and `Function<Serializable,C>`, which is illustrated by the second and third assignment that involve corresponding casts.

Despite of this compatibility, the first assignment fails. This is because there is no type information regarding the signature of the constructor reference `C::new` and the compiler must deduce all missing information from the left-hand side target type. For this purpose the compiler first replaces the wildcard `? super String` used on the left-hand side target type by its lower bound `String`. The resulting target type is `Function<String,C>`. With this target type information the compiler goes looking for a constructor in class `C` with one argument of type `String`.

Both overloaded constructors have a matching signature and the compiler considers `C::new` an ambiguous constructor reference.

Overloaded methods can create similar problems when used in lambda expressions. Here is the same example, this time using lambda expressions instead of constructor references:

```
Function<? super String,C> f;
f = x -> new C(x);                                    // error
f = (CharSequence x) -> new C(x);                     // fine
f = (Serializable x) -> new C(x);                     // fine
```

The compiler again considers `Function<String>` the target type and complains that both constructor of class `C` accept `String` as the argument type and are therefore ambiguous.

Wildcard Target Types Are the Norm

Part of the problem described above is due to the unspecific target type. In the example the target type is a wildcard parameterization of a generic type. Such a wildcard type stands for an entire family of types, not just one specific type. This increases the chance that several candidates from a set of overloaded methods meet the requirements.

When the target type is more specific, an ambiguity is less likely or does not occur at all. For instance, the ambiguity vanishes if we assign the overloaded constructor reference to a concrete parameterization instead of a wildcard parameterization:

```
Function<Function<CharSequence,C> f1 = C::new;          // fine
Function<Serializable,C>) f2 = C::new;                  // fine
```

Here the compiler can draw enough information from the target type in order to rule out one of the two overloaded version as inapplicable for this specific context. In essence, the more relaxed and unspecific the target type is the more likely is an ambiguity error.

Unfortunately, wildcard parameterizations are fairly common in practice as target types. For instance, almost all stream operations (and many other JDK APIs) have wildcard argument types. As a result, the unspecific wildcard target type is more of the norm rather than the exception. Below is an example of an invocation context that leads to the same ambiguity that we previously encountered in an assignment context.

Example of ambiguous constructor reference in an invocation context:

```
Arrays.asList("abc","xyz")
      .stream()
      .map(C::new)                                      // error
      .forEach(System.out::println);
```

The reference `C::new` to the overloaded constructor from our previous examples appears in a method invocation context, namely as the argument to the stream's `map` operation. The `map` operation's declared argument type is the wildcard type `Function<? super T,? extends R>`. In the given context it boils down to the required target type `Function<? super String, ? extends Object>`. This unspecific target type leads to the same ambiguity error message that we discussed above in the assignment context. Resolutions include casting the method reference or using a different type of stream.

Solving the problem with a cast:

```
Arrays.asList("abc","xyz")
      .stream()
      .map((Function<CharSequence,C>)C::new)
      .forEach(System.out::println);
```

We can add the same cast that we've been using in the assignment context and cast the constructor reference to `Function<CharSequence,C>`, which is no longer a wildcard type, but a concrete parameterization of the generic target type.

Alternatively, we can use a different stream type which alters the invocation context so that the map method has a different required argument type.

Solving the problem with an explicit type argument:

```
Arrays.<CharSequence>asList("abc","xyz")
      .stream()
      .map(C::new)
      .forEach(System.out::println);
```

We specify an explicit type argument for the generic `asList` method. This has the effect that the `asList` method returns a `List<CharSequence>` instead of a `List<String>`, which it returns without the explicit type argument. The stream is then a `Stream<CharSequence>` and its map operation has the declared argument type `Function<? super CharSequence,? extends Object>`. With this type information the compiler looks for a constructor that take `CharSequence` or a supertype therof as an argument, which prunes the candidate set to a single, unambiguous candidate.

More on Wildcard Target Types

Diesen Teil vielleicht benutzen, um die Type Inference für Wildcard Target Types zu erklären (Target Type is the Parameterization with the

Bound) - vielleicht kürzer und im regulären Abschnitt über Target Typing. Und ein realistischeres Beispiel nehmen:

Functional interfaces can be generic and they may be implemented by matching lambda expressions or method/constructor references. It means that target types may be (concrete or wildcard) parameterizations of generic types. Wildcard parameterization in particular can lead to type inference failures

Here is an example of a generic functional interface:

```
@FunctionalInterface
interface Factory<T> {
  Generic<T> make();
}
```

The functional interface `Factory` has the descriptor `<T>()->Generic<T>`, i.e., it has an unbounded type parameter, takes no arguments, does not throw checked exceptions, and returns a parameterization of a generic class named `Generic`.

Method/Constructor References & Wildcard Target Types

The generic `Factory` interface can be implemented by a constructor reference that refers to the default constructor of class `Generic`:

```
Factory<?> f1 = Generic::new;                                    //1
Factory<?> f2 = Generic<Object>::new;                            //2
Factory<?> f3 = Generic<String>::new;          // error          //3
Factory<?> f4 = (Factory<String>)Generic<String>::new;           //4
```

The implementation of the generic functional interface `Factory` in line `//1` is via the constructor reference `Generic::new`. `Generic` is a generic class and for this reason the function descriptor of its constructor is generic, namely `<T>()->Generic<T>`. Since we did not specify a type parameter that would replace the unknown type `T`, the compiler must deduce the type parameter. It does so by taking a look at the left-hand side of the assignment. There it finds the wildcard parameterization `Factory<?>`, which imposes no requirements regarding the type parameter, but also does not provide any information for its inference. The compiler then replaces the wildcard parameterization by a wildcard-free type for further type deduction and decides that the target type shall be `Factory<Object>`. (This is basically because the wildcard `'?'` does not have a bound; bounded wildcards would be replaced by their bound in this step of the type deduction.) The type `Factory<Object>` is a viable target type. It is compatible to the left-hand type `Factory<?>` and the constructor reference `Generic::new` can be converted to it.

Note, for the purpose of type inference the compiler treats the wildcard parameterization `Factory<?>` like `Factory<Object>` (or the raw type `Factory` for that matter).

The implementation in line `//2` is via the constructor reference `Generic<Object>::new`. This time the constructor reference is no longer generic. Instead it has the function descriptor `()-> Generic<Object>` and can be converted to the target type `Factory<Object>`.

The implementation in line `//3` is via the constructor reference `Generic<String>::new`. Its function descriptor is `()-> Generic<String>` which can be converted to `Factory<String>`, but not to `Factory<Object>`. Since the compiler treats the left-hand side type `Factory<?>` like `Factory<Object>` for the purpose of target typing it issues an error message.

In line `//4` we inserted a cast to `Factory<String>` which changed the inference context from an assignment context to a casting context. The relevant target type is now `Factory<String>`. The constructor reference `Generic<String>::new` has the function descriptor is `()->Generic<String>` which can be converted to the target type `Factory<String>`. Hence, target typing works and line `//4` compiles.

Lambda Expressions & Wildcard Target Types

Here are the same situations, this time using lambda expressions for implementation of the wildcard target type:

```
Factory<?> f1 = ()->new Generic<>();                          //1
Factory<?> f2 = ()->new Generic<Object>();                    //2
Factory<?> f3 = (Factory<String>) ()->new Generic<Object>();  //3
```

These work exactly like the constructor reference above. For the first lambda expression the compiler must infer the missing type parameter and infers `T:= Object`. The second lambda expression already has the matching function descriptor `()->Generic<Object>`. The third lambda expression is convertible to `Factory<String>` and requires an explicit cast.

Anonymous Inner Classes & Wildcard Target Types

Anonymous classes are treated differently regarding target typing. In contrast to lambda expressions and method/constructor references, an anonymous inner class definition is not a poly expression. It is a standalone expression and the compiler need not infer its type from the context.

Let us see how the generic target type would be implemented by anonymous inner classes:

```
Factory<?> f1 = new Factory<>() {  /*error*/                  //1
  public Generic<Object> make() { return new Generic<>(); }
};
Factory<?> f2 = new Factory<Object>() {                       //2
  public Generic<Object> make() { return new Generic<>(); }
};
Factory<?> f3 = new Factory<String>() {                       //3
  public Generic<String> make() { return new Generic<>(); }
};
```

The first attempt in line `//1`ff is rejected by the compiler because the diamond operator `'<>'` is not permitted for the supertype of an anonymous inner class. The supertype must be fully specified and must not require type inference.

The different treatment regarding type inference is also visible in the third attempt in line `//3`ff where the anonymous class implements `Factory<String>`. The corresponding implementations via a lambda expression or a constructor reference did not compiler without an explicit cast to `Factory<String>`. No such cast is needed for the anonymous class because it is not subject to type inference.

Target Types with a Generic Functional Method

Functional interfaces can have a single abstract method that is a generic method. Implementing such a functional interface requires that the implementation has a matching generic method. In principle, functional interfaces can be implemented by classes, lambda expressions, and method/ constructor references. The question is: can they provide a matching implementation of the required generic method? The answer is: classes and method/constructor reference can, but lambda expressions cannot.[7] Let us see why this is.

Here is an example of a functional interface with a generic method:

```
@FunctionalInterface
interface Factory {
  <T> Generic<T> make();
}
```

It uses a generic type named Generic that looks like this:

```
class Generic<X> {
  public Generic() { ... }
}
```

[7] Functional interfaces whose single abstract method is generic have already been mentioned in the section on "Generic Lambda Expressions Not Permitted".

The functional interface `Factory` itself is not generic, but its singe abstract method is a generic method. The method's descriptor is `<T>()->Generic<T>`, i.e., it has an unbounded type parameter, takes no arguments, does not throw checked exceptions, and returns a parameterization of `Generic`.

Here is how the target type can be implemented by a constructor reference:

```
Factory f1 = Generic::new;
Factory f2 = Generic<?>::new;          // error: illegal syntax
Factory f3 = Generic<Object>::new;     // error: incompatible
```

The first implementation of the functional interface `Factory` is via the constructor reference `Generic::new`. The compiler accepts it because it yields a generic function. This is because `Generic` is a generic class and its constructor's function descriptor is generic, namely `<T>()->Generic<T>`.

The second constructor reference `Generic<?>::new` is rejected because no wildcard types are permitted before the `'::'` symbol of a method/ constructor reference.

The third constructor reference `Generic<Object>::new` is a legal one, but it is incompatible to the target type `Factory`. The references `Generic<Object>::new` has the non-generic descriptor `()->Generic<Object>`, while the generic descriptor `<T>()->Generic<T>` is required.

Any attempt of implementing the target type `Factory` with its generic abstract method by means of a lambda expression is doomed to fail. Java does not support generic lambda expressions. In order illustrate the lack of generic lambda expressions let us try write a generic lambda expression.

Here are a couple of attempts to implement the target type's generic abstract method by a lambda expression:

```
Factory f1 = () -> new Generic<>();      // error: incompatible
Factory f2 = <T> () -> new Generic<T>(); // error: illegal syntax
```

The first lambda expression yields the non-generic function descriptor, namely a function that takes no arguments, does not throw, and returns some parameterization of `Generic`. It is incompatible to the generic method that the target type `Factory` requires.

The second lambda expression is simply illegal. There is no syntax for specification of type parameters for a lambda expression.

In essence, lambda expressions cannot implement functional interfaces with a generic method.

Anonymous inner classes, of course, can implement functional interfaces with a generic method. Here is an implementation of the target type by an anonymous inner class:

```
Factory f1 = new Factory() {
  public <T> Generic<T> make() { return new Generic<T>(); }
};
```

Classes have no restrictions regarding the methods that they implement and can easily provide a generic method if required.

Non-Abstract Methods in Interfaces

Traditionally, in Java all methods declared in an interface are abstract in the sense that an interface method just describes the signature of a method, but does not provide an implementation. The implementation has to be provided by a class that implements the interface and overrides the abstract methods.

- Since Java 8, interface methods can be non-abstract, i.e., they can provide an implementation. There are two types of non-abstract interface methods:
- default methods, and
- static methods.

A default method is a method with the modifier `default`. Its body provides a default implementation for any class that implements the interface without overriding the method. This allows new functionality to be added to existing (and perhaps already widely-distributed) interfaces without affecting any of the implementing classes.

An interface may also declare `static` methods. They work in much the same way as `static` methods in classes, except that they are not inherited. They can only be invoked via the interface, not by means of a subtype or an object.

An interface method that is neither `default` nor `static` is implicitly `abstract`.

Default Interface Methods

Default methods were added to the Java programming language in Java 8 in order to permit *interface evolution*.[8] Many of the existing JDK abstraction underwent a major overhaul for Java 8 and the library implementers had to modify these existing and widely used JDK classes for Java 8. Without default interface methods any modification of an existing interface had affected all implementing classes, i.e., a backward compatible modification had been impossible. This lack of support for interface evolution lead to the invention of default interface methods.

---

[8] This topic is also discussed in the *Lambda Tutorial* document (see the section on "Interface Evolution" in the *Lambda Tutorial* document).

For illustration we study an example of an interface that has been extended in JDK 8, namely the `Comparator` interface in package `java.util`. Before Java 8 it looked like this:

The Comparator interface before Java 8:

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
}
```

It has a `compare` method, which is an abstract method that subclasses must implement. In addition it has an `equals` method, which is not an abstract method, but a method inherited from class `Object` that subclasses need not implement.

Since Java 8 the `Comparator` interface has additional non-abstract methods, among them two default methods. They provide useful functionality implemented on top of the abstract `compare` method such as creation of a comparator for the reverse sorting order or composition of comparators.

Excerpt of the Comparator interface since Java 8:

```
public interface Comparator<T> {
    int compare(T t1, T t2);
    boolean equals(Object obj);
    default Comparator<T> reverseOrder() {
       return Collections.reverseOrder(this);
    }
    …
}
```

The other default method is the `thenComparing` method that exists in half a dozen overloaded versions.

Default methods are qualified by the modifier `default` and are implicitly `public`. A default method must have a body with an implementation. For the implementation of the `reversOrder` method the original comparator is used - referred to via the `this` keyword.

Here is an example of using the reverse comparator:

```
void testComparator() {
        String[] array =  {"a", "b", "c"};
        Comparator<String> cmp = (x,y)-> x.compareTo(y);
        Arrays.sort(array,cmp);
        System.out.println(Arrays.toString(array));
        Arrays.sort(array,cmp.reverseOrder());
        System.out.println(Arrays.toString(array));
}
```

The resulting output is:

```
[a, b, c]
```

```
[c, b, a]
```

For implementation of the default method `reverseOrder` we used the `this` reference and a static method from an unrelated class, namely the `reverseOrder` method from class `Collections`. For providing an implementation of a default method we can also use all the arguments passed to the method, if any, and all other methods defined in the interface. This is illustrated by the `thenComparing` method defined in the `Comparator` interface.

A more complete excerpt of the Comparator interface since Java 8:

```
public interface Comparator<T> {
    int compare(T t1, T t2);
    boolean equals(Object obj);
    default Comparator<T> reverseOrder() {
       return Collections.reverseOrder(this);
    }
    default
    Comparator<T> thenComparing(Comparator<? super T> other) {
        Objects.requireNonNull(other);
        return (Comparator<T> & Serializable) (c1, c2) -> {
            int res = compare(c1, c2);
            return (res != 0) ? res : other.compare(c1, c2);
        };
    }
    …
}
```

It uses the abstract `compare` method for its implementation.

Basically, a default interface method is implemented on top of the rest of the interface, i.e., by means of the abstract and non-abstract methods defined in the interface. A default method typically provides functionality that is a combination or adaptation of the abstract interface methods.

One restriction (compared to non-abstract methods in classes) remains: interfaces still do not have data. Interfaces may define constants, i.e., final fields with compile-time constant values, but interfaces must not define regular, mutable, non-final fields (like a class may do).

Modifiers - Permitted and Prohibited

Default interface methods are implicitly `public`; they can neither be `protected`, `private`, nor package visible. There is no compelling reason for this restriction regarding the accessiblity modifiers. Non-public accessibility is rarely needed and there were simply more important issues to care about. It is, however, conceivable that future versions of Java may allow the full set of accessiblity modifiers.

Default interface methods must not be `abstract`, which is obvious since a default method is expressly meant to be overridden by a subclass.

Default interface methods must not be `static`. This, too, is sensible because default interface methods are non-static methods that are inherited into subtypes and may be overridded by subclasses.

Default interface methods must not be `final`. There are two reasons for this. The first reason is that is was expressly intended that default interface methods should be overridable. Every subclass should be allowed to provide an alternative implementation for the default implementation offered by the interface.

The second reason is interface evolution. When a default method is added to an interface then it might have the same name as an existing method in an existing class that already implements the interface. If the additional interface method is non-`final` then the existing class method simply overrides the default interface method and no harm is done. If, in contrast, the default interface method were `final`, then the existing class would no longer compile because it illegally attempts to override a `final` method. So, for reasons of backward compatibility a default interface method cannot be declared `final`.

Multiple Inheritance

In the initial design of the Java language, interfaces were intended for the abstract description of a concept, i.e., they had no data and no functionality. Since Java 8 interfaces can have default methods and for this reason can provide pieces of implementation, which begs the question whether interfaces are pure API descriptions any longer.

An interface with default methods clearly does not meet the criteria of a pure API description any longer because it does have functionality. Is it a problem?

The answer to this question is related to multiple inheritance, which is a language feature that was deliberately restricted in Java to multiple inheritance of (purely abstract) interfaces. When Java was invented, multiple inheritance in general was considered "to bring more grief than benefit".[9] For this reason, the language designers allowed multiple inheritance only for interfaces and restricted it to single inheritance for classes.

---

[9] Quoted from "Java: an Overview" by James Gosling, February 1995, http://www.cs.dartmouth.edu/~mckeeman/cs118/references/OriginalJavaWhitepaper.pdf.

The Deadly Diamond of Death

The misgivings regarding multiple inheritance stem from programming languages that do have multiple inheritance of classes, most prominently C++. The problem of multiple inheritance occurs with a diamond shaped inheritance, sometimes refer to the *deadly diamond of death*. It is a situation where two types B and C are subtypes of another type A and then there is a type D that is a subtype of both type B and C.

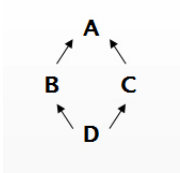


Diagram: Multiple Inheritance - The "deadly diamond of death"

Exactly this type of multiple inheritance causes trouble, if type A has state. A's state is inherited by its subtypes. An obvious question pops up: does D have one or two A-parts? After all, B inherits an A-part and C inherits an A-part. Should D then have two A-parts? Or perhaps just one? The C++ programming language permits the diamond shaped multiple inheritance among classes with data members and, to boot, offers both choices: it has virtual and non-virtual multiple inheritance, which leaves the decision regarding one or tow A-parts to the programmer. Making this decision already creates headache. Then there is the issue of "Who initializes the A-part, if there is only one A-part? B, C, or D?" Plus, there are tons of ambiguities with multiple inheritance. For instance, what does it mean if A, B, and C have a overlapping methods with the same name and signature and we invoke the method on a reference of type D? How does the compiler resolve the method call?

Fortunately, the most nasty multiple inheritance situation, namely the "deadly diamond of death" illustrated above is no issue in Java, not even with default methods in Java 8. Java remains restricted to single inheritance of classes and permits multiple inheritance only for interfaces. For this reason, type A in the diamond shaped inheritance must be an interface in Java; it cannot be a class. This is because type D is derived from type B and C. A subclass (like D) can only have one direct superclass, i.e., at most one of the two types B or C can be class; the other one must be an interface. Because type A is the supertype of at an interface (either B and/or C) it must be an interface, too.

If type A in "deadly diamond of death" is an interface then there is no question regarding how many A-parts type D will have. Type A does not

have state and hence there is no such thing as an A-part neither in B, C, or D.

Programming with Default Methods

Default methods were invented primarily for interface evolution, namely painless extension of interfaces that already have implementing subclasses. In practice, default methods can be used for many other things. They have the potential for changing the way in which we go about developing APIs.

Traditionally, we go about the business of developing an API in several steps:

- *Step 1: Interfaces.* First, we describe a new API as a pure abstraction by defining an interface without any implementations.
- *Step 2: Abstract Classes.* In a subsequent step we provide partial implementations of the interface.
- *Step 3: Concrete Classes.* Of these abstract classes we derive further classes that eventually are complete and no longer abstract.

Now, that we have default methods we still describe a new API by declaring a bunch of interface methods without any implementation. But the immediate next step might be definition of default methods that combine the yet abstract interface methods to useful additional functionality. Only then would we start providing actual implementations in terms of abstract and concrete classes. This way, part of what we used to do in step 2 now becomes part of step 1.

Since Java 8, we may choose to develop an API this way:

- *Step 1a: Abstract Interface Methods.* First, we describe a new API as a pure abstraction by defining an interface without any implementations.
- *Step 1b: Default Interface Methods.* Then, we add default methods that are defined on top of the abstract interface methods.
- *Step 2: Abstract Classes.* In a subsequent step we provide implementations for some of the abstract interface methods in a (potentially abstract) subclass.
- *Step 3: Concrete Classes.* Of these abstract classes we derive further classes that eventually are complete and no longer abstract.

Let us explore a couple of examples of the usefulness of default methods.

Example #1: Genuine Default Functionality

The `Iterator` interface from package `java.util` requires three methods: `hasNext`, `next`, and `remove`. In many iterator implementations the `remove`

method does not make sense and remains unsupported. Yet, the implementing class must provide an implementation of the `remove` method. Usually it just throws an `UnsupportedOperationException`.

Since Java 8 the Iterator interface has a default implementation of the `remove` method. Here is an excerpt of the `Iterator` interface:

```
public interface Iterator<E> {
  boolean hasNext();
  E next();
  default void remove() {
    throw new UnsupportedOperationException("remove");
  }
  …
}
```

In this example the default method is used to provide an actual default. It frees subclasses from the burden of implementing an abstract interface method that thay do not intend to support.

Example #2: Orthogonal Functionality

Consider the `Comsumer` interface in package `java.util.function` of the JDK. It is the interface that is used in conjunction with the stream's `forEach` method. It declares a single `accept` method. On top of this not yet implemented `accept` method one can already provide useful functionality such as creating a chain of two consumers.

```
@FunctionalInterface
public interface Consumer<T> {
  public void accept(T t);

  public default Consumer<T> andThen(Consumer<? super T> other){
        return (T t) -> { accept(t); other.accept(t); };
  }
}
```

Here is an example of using a chain of consumers for printing the elements of a sequence to two output channels:

```
String[] names = {"Eric", "Emma", "Eleanor"};
Arrays.stream(names).forEach(
   ((Consumer<String>)System.out::println)
   .andThen(System.err::println)
);
```

The `forEach` method then prints the strings to both `System.out` and `System.err` in one pass over the sequence.

In this example the default method is used to provide an orthogonal, additional piece of functionality, namely a factory for creation of a new `Consumer`.

Example #3: Convenience Functionality

The `Stream` interface, too, illustrates how helpful default methods can be. Similar to the `Collection` interface the `Stream` interface has two `toArray` methods:

```
public interface Collection<E>  {
    ...
    <T>  T[] toArray(T[] a);
    Object[] toArray();
    ...
}

public interface Stream<T>  {
  <A> A[] toArray(IntFunction<A[]> generator);

  default Object[] toArray() {
    IntFunction<T[]> generator = s -> (T[]) new Object[s];
    return toArray(generator);
  }
}
```

While the `Collection` interface forces each implementing class to implement both `toArray` methods the `Stream` interface requires only one implementation and supplies the second `toArray` method as a default method implemented based on the first one.

In this example the default method is used to provide a convenience method that is a slight variation of an existing method. This is different from the previous example where the default method provided orthogonal functionality.

Example #4: Adapter Functionality

Default methods can be used for retrofitting. Remember the `Enumeration` interface that came as part of JDK 1.0 and is for instance used by class `Vector`. It was later superseded in Java 1.2 by the `Iterator` interface. The enumeration is similar to an iterator, but its methods have different names. With default methods it is easy to have the `Enumeration` interface extend the `Iterator` interface.

```
interface Enumeration<E> extends Iterator<E> {
  boolean hasMoreElements();
  E nextElement();

  default boolean hasNext() { return hasMoreElements(); }
  default E       next()    { return nextElement(); }
  default void    remove()  { throw new
                                  UnsupportedOperationException();
                              }
}
```

With this simple extension of the `Enumeration` interface, every enumeration could serve as an iterator. The retrofitting suggested above is a hypothetical one.

The JDK does not provide this kind retrofitting because it is not needed. Class `Vector` itself was retrofitted and implements both the `Enumeration` and the `Iterator` interface. However, the enhanced Enumeration interface illustrates how default methods can be used for adaptations and retrofittings.

In this example the default methods serve as an adapter that maps the methods of one interface to the methods of another interface.

Example #5: Distinction From (Abstract) Classes

Now that interfaces can supply functionality in form of default methods, do we still need abstract classes or are they obsolete? It turns out that (abstract) classes are still needed. Let us take a look at a situation in which interfaces do not suffice to solve a given problem.

Consider the following interface:

```
interface Name {
  String getFirstName();
  String getMiddleName();
  String getLastName();
}
```

and its implementing class:

```
class NamedPerson implements Name {
  private String firstName;
  private String middleName;
  private String lastName;

  public NamedPerson(String first, String middle, String last) {
    firstName  = first;
    middleName = middle;
    lastName   = last;
  }
  public String getFirstName() {
    return firstName;
  }
  public String getMiddleName() {
    return middleName;
  }
  public String getLastName() {
    return lastName;
  }
  public String getName() {
    return firstName
         +(middleName!=null&&middleName.length()>0?
                                " "+middleName.charAt(0)+'.':"")
         +" "+lastName;
  }
  public String toString() {
    return String.format("%-12s= %s\n%-12s= %s\n%-12s= %s"
      ,"firstName" ,firstName
      ,"middleName",middleName
```

```
        ,"lastName"  ,lastName
      );
    }
  }
```

This is how we would split the API into an interface and a class traditionally, i.e., without default method. Using default methods we can provide the `getName` method in the interface already, because it is a mere convenience method that can be implemented on top of the three abstract interface methods.

After moving the `getName` method from the class to the interface it looks like this:

```
interface Name {
  String getFirstName();
  String getMiddleName();
  String getLastName();

  default String getName() {
    return getFirstName()
        +((getMiddleName()!=null&&getMiddleName().length()>0)?
                          " "+getMiddleName().charAt(0)+'.':"")
        +" "+getLastName();
  }
}

class NamedPerson implements Name {
  private String firstName;
  private String middleName;
  private String lastName;


  public NamedPerson(String first, String middle, String last) {
    firstName  = first;
    middleName = middle;
    lastName   = last;
  }
  public String getFirstName() {
    return firstName;
  }
  public String getMiddleName() {
    return middleName;
  }
  public String getLastName() {
    return lastName;
  }
  public String toString() {
    return String.format("%-12s= %s\n%-12s= %s\n%-12s= %s"
      ,"firstName" ,firstName
      ,"middleName",middleName
      ,"lastName"  ,lastName
    );
  }
}
```

Basically, we can implement all methods as default interface method that can be built on top of the abstract inferface methods.

Following this line of logic one might want to implement the `toString` method as a default interface method, too. This, however, is not permitted, because `toString` already has an implementation in the `Object` superclass. Any class that implements the `Name` interface would inherit two implementations of the `toString` method. The method defined in class `Object` would always win and the default interface method would always be ignored. In other words, defining a default interface method `toString` in an interface is pointless and for this reason prevented by the compiler right away.[10]

The three getter methods cannot be implemented as default interface method because they need access to data and interfaces cannot store data. This means that all methods that need access to data stored in fields must be implemented as class methods.

The example illustrates that default method allow implementation of methods that need no data and are typically combinations of the abstract interface methods. All methods that need data access must be implemented in abstract or concrete classes.

Ambiguities Involving Default Interface Methods

As both interfaces and classes supply non-abstract methods to their subclasses, the same method can be inherited from different supertypes. This can lead to conflicts. For illustration, here are a couple of ambiguity examples:

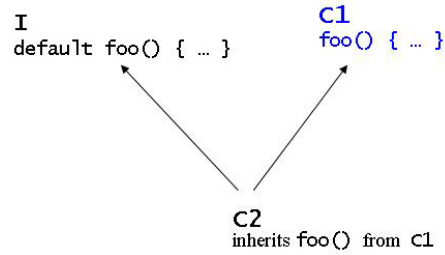
Ambiguity #1:

A class `C2` inherits a method `foo` from both an interface `I` and a class `C1`. Which method does subclass `C2` inherit?

Example: Ambiguous Multiple Inheritance - Class wins over interface.

[10] More on conflicts and ambiguities caused by default interface methods can be found in the section on "Ambiguities Involving Default Interface Methods" and "Ambuity #7" in particular.

```
interface I {
  default void foo() {...}
}
class C1 implements I {
  public void foo() {...}
}
class C2 extends C1 {}
```



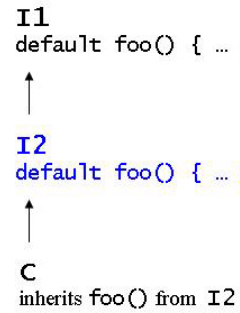
The simple rule is: the class wins. The `foo` method present in subclass `C2` is the one inherited from superclass `C1`. This behaviour reflects the idea of default methods: default methods are a fallback if the class hierarchy doesn't provide anything.

Ambiguity #2:

A class `C` inherits a method `foo` from an interface `I1` which inherits the same method `foo` from its superinterface `I2`. Which method does class `C` inherit?

Example: Ambiguous Multiple Inheritance - Closest super interface wins.

```
interface I1 {
  default void foo() { ... }
}
interface I2 extends I1 {
  default void foo() { ... }
}
class C implements I2 {}
```



The rule is: the closest super interface wins. The `foo` method present in subclass `C` is the one inherited from interface `I2`.

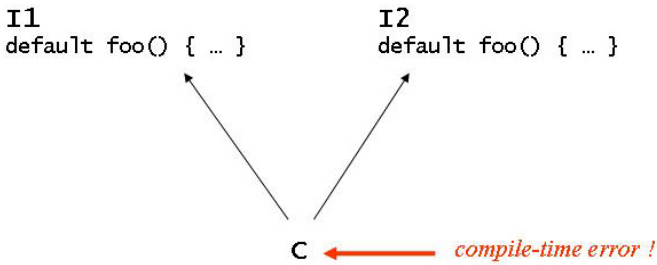
Ambiguity #3:

A class `C` inherits a method `foo` from both an interface `I1` and an interface `I2`. This time the two interfaces are not derived from each other and there is no closest interface. Which method does class `C` inherit?

Example: Ambiguous Multiple Inheritance - Compile-time error; needs explicit resolution.

```
interface I1 {
  default void foo() { ... }
}
interface I2 {
  default void foo() { ... }
```

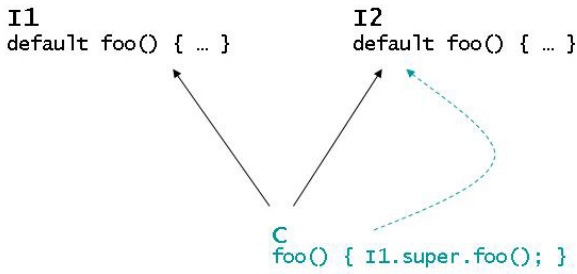
```
}
class C implements I1, I2 {}  // error
```



The compiler cannot resolve it and reports an error. The situation can be resolved by explicitly stating which method class `C` is supposed to inherit. A resolution could look like this:

Example: Disambiguation via `interface.super` in a subclass.

```
interface I1 {
  default void foo() { … }
}
interface I2 {
  default void foo() { … }
}
class C implements I1, I2 {
  public void foo() {
    I2.super.foo();
  }
}
```



The method invocation via `interface.super` is not restricted to methods of classes; it can also be used in default methods of interfaces. Here is an example:

Example: Disambiguation via `interface.super` in an subinterface.

```
interface I1 {
  default void f() { … }
}
interface I2 {
  default void f() { …}
}
interface I3 extends I1, I2 {
```

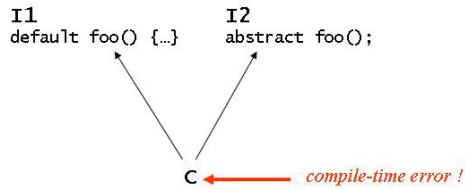
```
  default void f() {
    I1.super.f();
  }
}
```

Ambiguity #4:

A class `C` inherits a method `foo` from both an interface `I1` and an interface `I2`. The method is abstract in one interface and has a default implementaion in the other interface. Is method `foo` abstract or default in class `C`?

Example: Ambiguous Multiple Inheritance - Compiler-time error; needs explicit resolution.

```
interface I1 {
  default void foo()
  { ... }
}
interface I2
{  void foo();  }
class C implements I1, I2
{}  // error
```



The compiler cannot resolve it and reports an error. The situation can be resolved by explicitly declaring the method as abstract or by implementing it. A resolution would look like this:

```
class C implements I1, I2 {
   public void foo() { I2.super.foo(); }
}
```

or like this:

```
abstract class C implements I1, I2 {
   public abstract void foo();
}
```

Ambiguity #5:

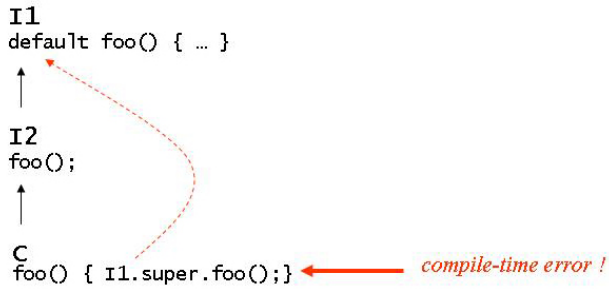
A class `C` inherits from an super-interface `I2` which inherits from a super-super-interface `I1`. The topmost interface `I1` has a default version of a method `foo`. The direct interface `I2` declares the same method `foo` as abstract. Class `C` inherits the closest version of method `foo`, which is the abstract one from interface `I2`. What if class `C` wants to use the grandparents default method?

The following is illegal because the interface preceding the `super` keyword must be a *direct superinterface*.

Example: No access to super-super-interface's default method.

```
interface I1 {
  default void foo() { ... }
```

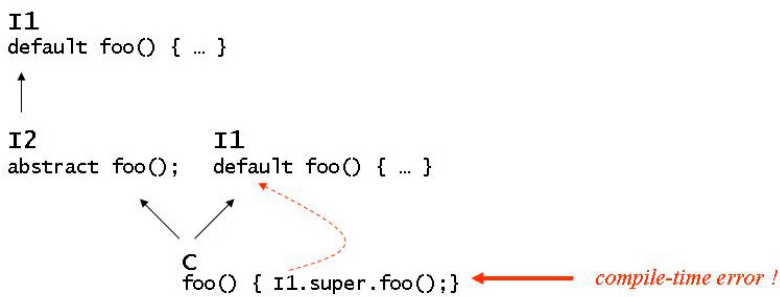
```
}
interface I2 extends I1 {
  void foo();
}
class C implements I2 {
  public void foo() {
    I1.super.foo();  // error
  }
}
```



It is illegal to skip the direct interface. The following is illegal, too. It is the attempt to get access to the super-super-interface's default method by simply repeating it as direct interface.

Example: Cannot skip direct interface.

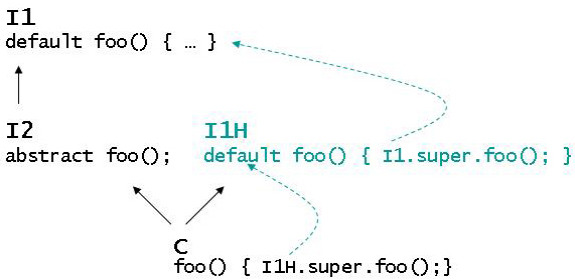
```
interface I1 {
  default void foo() { ... }
}
interface I2 extends I1 {
  void foo();
}
class C implements I1, I2 {
  public void foo() {
    I1.super.foo();  // error
  }
}
```



As a work-around we can define an additional interface as a helper that exposes the desired behaviour with a `super` method invocation. Here is the helper interface:

Example: Work around via helper interface.

```
interface I1 {
  default void foo() { ... }
}
interface I2 extends I1 {
  void foo();
}
interface I1H extends I1 {
  default void foo() {
    I1.super.foo();
  }
}
class C implements I1H, I2 {
  public void foo() {
    I1H.super.foo();  // fine
  }
}
```



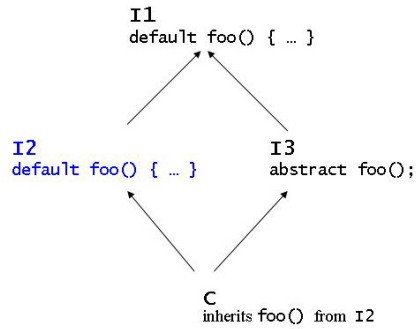
Ambiguity #6:

A class `C` inherits a method `foo` from both two interfaces `I2` and `I3` that share a common ancestor `I1`. One of the direct superinterfaces overrides the method, the other does not. . Which method does class `C` inherit?

Example: Ambiguous Multiple Inheritance - The closest method wins.

```
interface I1 {
  default void foo() { ... }
}
interface I2 extends I2 {
  default void foo() { ... }
}

interface I3 extends I1 {
  void foo();
}
class C implements I2, I3 {}
```



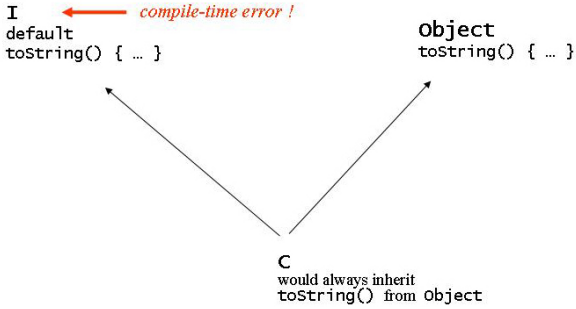
Again, the closest method wins. Methods (like `I1::foo`) that are already overridden by other candidates (by `I2::foo` in the example) are ignored.

Ambuity #7:

An interface defines methods from class `Object` as default methods. For instance, an interface might attempt to provide default implementations of the `hashCode`, `equals`, or `toString` method.

Example: Default Methods for Public Methods from Class `Object`

```
interface I {
  default String toString() { … }              // error
  default boolean equals(Object other) { … }   // error
  default int hashCode() { … }                 // error
}
```

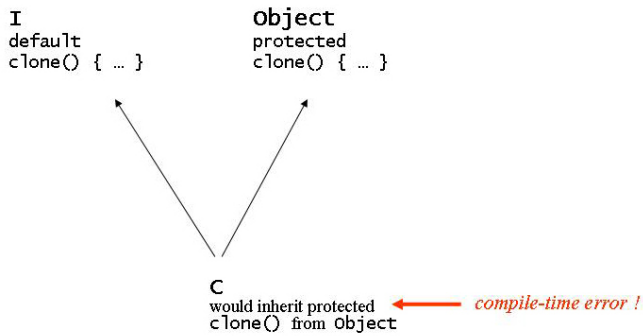


Providing default interface methods that collide with public methods from class `Object` is not permitted. The reason is that a class that implements the interface would inherit two versions of the method, the one implemented in class `Object` and the one implemented in the interface. The method from class `Object` would always win (see Ambiguity #1). Hence it is pointless to provide default implementations of `hashCode`, `equals`, or `toString` and the compiler prevents it right away.

It is permitted to provide a default implementation of the `clone` method from class `Object`. It is, however, debatable.

Example: Default Method for Protected `clone` Method from Class `Object`

```
interface I extends Cloneable {
  default I clone() { … }
}
class C implements I { … }  // error
```

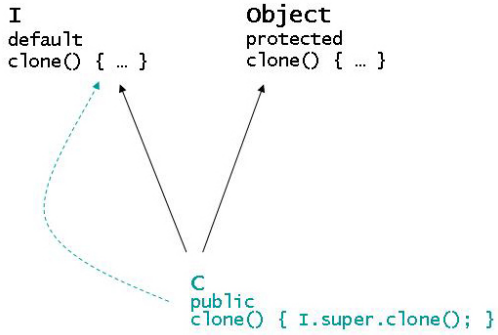


The compiler rejects class `C` because it inherits the protected `clone` method from class `Object` and the public default method from interface `I`. Since the class's method wins it is an attempti to assign weaker access privileges to the `clone` method: it was public in a supertype (interface `I`) and is now protected in class `C`.

All classes can, of course, override the interface's default `clone` method, which renders the default implementation entirely pointless. Interestingly, an overriding method in the class can refer to the default method defined in the interface.

Example: Default Implementation of `clone` is Overridden and Used:

```
interface I extends Cloneable {
  default I clone() { … }
}
class C implements I {
  return (C)I.super.clone();
}
```



The approach is debatable though. It is hard to imagine that the interface can provide a reasonable default implementation of a `clone` method. Cloning is usually about copying data stored in the instance fields and the interface does not have access to any instance field. Situations where a default `clone` method in an interface is useful will probably be rare.

The list presented here of examples of ambiguous multiple inheritance is by no means comprehensive. Many more ambiguities can occur. In practice they are harmless. There is a resolution rule for most situations and in those few cases where the compiler cannot resolve the ambiguity there is syntax for explicit resolution.

Static Interface Methods

Since Java 8, interfaces may define static methods. Like default methods they must have an implementation. Static methods were added to the language because occasionally an interface is the most appropriate place to declare methods that produce or manipulate objects of the interface type.

Here is an example of an interface with a static method. It is the `Predicate` interface from package `java.util.function`:

Example of a static interface method:

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
    …
}
```

The static interface method `isEqual` is a factory method that creates a new predicate which yields `true` if the tested value is equal to a particular value. Here is an example of using it:

```
Predicate<Integer> isFive = Predicate.isEqual(5);
Stream.of(0,1,2,3,4,5,6,7,8,9)
      .filter(isFive)
      .forEach(System.out::println);
```

In the example we create a predicate names `isFive` that test whether a value is equals to `5`.

Static vs. Default Interface Methods

How do default interface methods differ from static interface methods? Both are non-abstract and have implementations. The difference is that default methods can access all members of the interface whereas static methods may only access static members. Also, a default method in a generic interface has access to the interface's type variables whereas a static method has no access to the type variables. It is exactly the same distinction as between static and non-static methods in classes.

The `Predicate` interface illustrates the difference. It has default methods in addition to the abstract and static method.

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

  static <T> Predicate<T> isEqual(Object targetRef) {
    return (null == targetRef)
           ? Objects::isNull
           : object -> targetRef.equals(object);
  }
  default Predicate<T> negate() {
    return (t) -> !test(t);
  }
  default Predicate<T> and(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) && other.test(t);
  }
  default Predicate<T> or(Predicate<? super T> other) {
    Objects.requireNonNull(other);
    return (t) -> test(t) || other.test(t);
  }
}
```

The static method `isEqual` does not invoke any of the interface's methods; in particular it does not access any non-static members of the `Predicate` interface. In contrast, the default method `negate` calls the abstract method `test`. It must be non-static.

The default method `negate` is a factory method, too. Here is an example of using it:

```
Predicate<Integer> isNotFive = isFive.negate();
Stream.of(0,1,2,3,4,5,6,7,8,9)
      .filter(isNotFive)
      .forEach(System.out::println);
```

The default method `negate` creates a new predicate that is an adapter of an existing predicate. It yields `true` when the adaptee returns `false` and `false` when the adaptee returns `true`.

More generally, static interface methods can invoke other static interface methods and have access to compile-time constant fields defined in the interface, but they cannot call abstract and default methods. Here is an example:

```
interface I {
  int CONSTANT = 42;
  abstract void abstractMethod();
  default  void defaultMethod1() {}
  static   void staticMethod1() {}

  default void defaultMethod2() {
    int i = CONSTANT;
    abstractMethod();
    defaultMethod1();
    staticMethod1();
  }
  static void staticMethod2() {
    int i = CONSTANT;
    abstractMethod();              // error
    defaultMethod1();              // error
    staticMethod1();
  }
}
```

The access to constant values is because they are implicity `public`, `static`, and `final`. Abstract and default methods, in contrast, are implicitly `public`.

Similarly, static interface methods have no access to type variables of the enclosing interface, but they do have access to nested types. Here is an example:

```
interface I<T> {
  interface NestedInterface {}
  class     NestedClass {}
  enum      NestedEnum {ENUM}

  default  void defaultMethod() {
    T t;
    NestedInterface x;
    NestedClass     y = new NestedClass();
    NestedEnum      z = NestedEnum.ENUM;
  }
  static   void staticMethod() {
    T t;                                  // error
    NestedInterface x;
    NestedClass y = new NestedClass();
    NestedEnum  z = NestedEnum.ENUM;
  }
}
```

Again, this is because nested types defined in an interface are implicitly `static`.

Modifiers - Permitted and Prohibited

Static interface methods are implicitly `public`; they can neither be declared `protected`, nor `private`, nor package visible. There is no compelling reason for this restriction regarding the accessiblity modifiers. Non-public accessibility is rarely needed and there were simply more important issues to care about. It is, however, conceivable that future versions of Java may allow the full set of accessiblity modifiers.

Static interface methods must not be `abstract`. This is in line with the same rule for static class methods.

Static interface methods must not be qualified as `default` method. Default methods are meant as non-static methods. The combination of the modifiers `static` and `default` is illegal.

Static interface methods must not be `final`. This is different from static methods in classes. For a static class method the `final` qualifier prevents redefinition of the method in a subclass. For a static interface method the `final` qualification is not needed because static interface methods cannot be overridden; they must be invoked via their exact type - as we will see in the next section.

Static Interface vs. Static Class Methods

We can implement static methods in interfaces and we can implement static methods in classes. How do they differ?

Static interface methods differ from static class method in three aspects:

- They cannot be invoked via an instance.
- They can only be invoked via the interface type in which they are declared.
- They are not inherited.

A static class method can be invoked either using the class name or using a reference to an object. A static interface method must be called via the interface name; using an object reference is not allowed for invocation of a static interface method. Here is an example:

```
interface I {
  public static void f() { … }
}
class C implements I {
  public static void f() { … }
}
```

```
public static void main(String... args) {
  I ir = new C();
  C cr = new C();
  ir.f();             // error
  cr.f();             // fine
  I.f();
  C.f();
}
```

The example shows that the static interface method cannot be called using an object reference.

A static class method can be invoked via the class type in which the static method is defined or via any subclass thereof. A static interface method must be called via the interface type in which the static method is defined; using a subtype for for invocation of a static interface method is not permitted. Here is an example:

```
interface I1 {
  public static void f() { … }
}
interface I2 extends I1 {
}
class C1 implements I2 {
  public static void f() { … }
}
class C2 extends C1 implements I2 {
}

public static void main(String... args) {
  I1 ir1 = new C2();
  I2 ir2 = new C2();
  C1 cr1 = new C2();
  C2 cr2 = new C2();
  I1.f();
  I2.f();         // error
  C1.f();
  C2.f();         // fine
}
```

The example illustrates that a derived interface cannot be used for invocation of a static interface method. Essentially it means that static class methods are inherited, but static interface methods are not.

Static Import for Interface Methods

Static interface methods can be imported. There is no difference compared to static class methods. Here is an example:

```
package package1;
public interface I {
  public static void f() { … }
}
class C implements I {
  public static void g() { … }
```

```
}

package package2;

import static package1.I.*;
import static package1.C.*;

public static void main(String... args) {
  f();
  g();
}
```

Note that a static class method with out an accessibility qualifier cannot be imported because it is only package visible, whereas a static interface method without an accessibility qualifier is `public` by default and can be imported.

Inheritance of Static Methods

Static methods in interfaces look like static methods in classes and work in much the same way, except that they are not inherited. Inheritance of static interface methods is deliberately disallowed in order to prevent that modification of an interface by adding a static method breaks the code of existing subclasses. In general, the goal is that non-abstract methods (i.e. static and default methods) can be added to an interface without affecting the interface's subclasses.

This is different for static methods in classes. It has never been a goal to prevent that modification of a superclass has no effect on its subclasses. For this reason inheritance of static class methods is permitted and it is accepted that adding a static method to a superclass can break the subclasses' code.

For illustration of the difference between inheritance of static methods in interfaces and classes we will study two type hierarchies in which we will modify the topmost supertype by adding a static method. First the topmost supertype is a class. Then we consider a situation where the topmost supertype is an interface.

First we consider an example with classes. Here is the initial class hierarchy:

```
interface SuperInterface {
}
class SuperSuperClass  {
}
class SuperClass extends SuperSuperClass
      implements SuperInterface {
  public static void method(SuperClass arg) { ... }
}
class SubClass extends SuperClass {
}
```

When we invoke the method as follows:

```
SuperClass.method(new SubClass());
                          // calls SuperClass::method(SuperClass)
```

the expected happens: the `SuperClass`'s method is invoked and its argument of type `SubClass` is converted to the required parameter type `SuperClass`.

Now we modify the superclass and add a static method with the same name but a different argument type to the `SuperSuperClass`.

```
interface SuperInterface {
}
class SuperSuperClass  {
  public static void method(SubClass arg) { ... }
}
class SuperClass extends SuperSuperClass
      implements SuperInterface {
  public static void method(SuperClass arg) { ... }
}
class SubClass extends SuperClass {
}
```

We do not change the method invocation; it is exactly the same as above:

```
SuperClass.method(new SubClass());
               // now calls SuperSuperClass::method(SubClass) !!!
```

Yet another method is called, namely the new one from the `SuperSuperClass` whose declared parameter type perfectly matches the argument type so that no conversion is required.

This perhaps unexpected effect occurs because static methods in classes are inherited. After adding a static method to the superclass the compiler can choose between two candidate methods and picks the better match. The same would happen if static methods in interfaces were inherited. Adding a static method to an interface could silently change the meaning of existing code. Since default and static interface methods were intended for interface evolution, any side effects on existing code is undesired and for this reason static interface method are not inherited.

In order to see the difference we now added a static method to the interface instead of the superclass:

```
interface SuperInterface {
  public static void method(SubClass arg) { ... }
}
class SuperSuperClass  {
}
class SuperClass extends SuperSuperClass
      implements SuperInterface {
  public static void method(SuperClass arg) { ... }
}
class SubClass extends SuperClass {
}
```

We do not change the method invocation; it is exactly the same as above:

```
SuperClass.method(new SubClass());
                    // still calls SuperClass::method(SuperClass)
```

This time the added static method does not change the meaning of the method invocation. The static interface method is not inherited, which means it cannot be qualified by a subtype's name. The only type qualifier permitted for the static interface method is the name of the declaring interface. That is, `SuperClass.method` and `SubClass.method` refer to the static class method, whereas `SuperInterface.method` is the only permitted reference to the static interface method.

Programming with Static Interface Methods

How the Java community will ultimately be using static interfaces methods remains to be seen. In the following we take a look at the usage of static interface methods in the JDK:

Example #1: Substitution of Companion Classes

In a library like the JDK there are situations where functionality is closely related to an abstraction that is expressed as an interface. An example is the `Collection` interface in package `java.util`. It describes the operations that are common to all collections such as `add`, `remove`, `contains`, `size`, `clear`, etc. Concrete subclasses, such as `ArrayList` and `HashSet`, provide implementations for this interface. Algorithms that perform useful functions on collections, such as `max`, `min`, or `sort` for instance, are implemented separately as static methods in a companion class named `Collections`.

Such pairs of an interface and a companion class with static support methods are fairly common. The pair `Collection` / `Collections` is one example. There are other examples, for instance `Executor` / `Executors` in package `java.util.concurrent`, `Channel` / `Channels` in package `java.nio`, `Path` / `Paths` in package `java.nio.file`. The companion class is usually, but not always, named after the corresponding interface and uses the plural of the interface's name as its name.

In principle, one could now get rid of all the companion classes by moving the companion classes' static methods to the related interface.

Collection / Collections

In the case of the `Collection` / `Collections` pair it turns out that the `Collections` class has so many static methods (~60 methods) that their addition to the `Collection` interface would overwhelm the comparatively

slim interface (~20 methods). To boot, class `Collections` has methods related to other interface such as `List` and `Set`, for instance adapter methods such as `synchronizedList` and `synchronizedSet`, which could or should be relocated to the `List` or `Set` interface instead of the `Collection` interface.

So, for historic reasons and for sake of backward compatibility, the `Collection` / `Collections` pair will not be unified by means of static interface methods.

Let us take a look at more recently designed abstractions such as streams and collectors for instance.

Stream / Streams

Package `java.util.stream` has an interface `Stream` with many non-static methods (~ 40 methods) and a handful of static method (6 methods). Most static method in interface `Stream` are factory methods that create new streams like for instance

```
static <T> Stream<T> empty()
static <T> Stream<T> of(T... values)
static <T> Stream<T> generate(Supplier<T> s)
…
```

Initially, these static factory methods were located in a separate companion class `Streams`, but eventually the decision was that they were too few to deserve a class of their own.

Collector / Collectors

Another example is the pair `Collector` / `Collectors` in package `java.util.stream`. The companion class `Collectors` has more that 30 static methods, whereas the `Collector` interface has only 5 abstract methods. Adding ~30 static methods to an interface with just 5 abstract methods renders the interface unreadable. After all the key feature of an interface is its abstracts methods and they should not disappear in an abundance of static methods.

Note that the previous case of `Stream` / `Streams` was an example of the opposite: adding 6 static methods to an interface with ~40 abstract methods does not impair the interface's readability As long as the abstract methods outnumbers the static methods the interface is still recognizable as an abstraction.

Example #2: Factory Methods

Interestingly, not all static methods related to the `Collector` interface have been placed into the companion class `Collectors`. Two (out of ~ 30) static methods are defined in the `Collector` interface, which bears the question why they are not located in the `Collectors` class along with all the other static method.

The reason is that the two static interface methods are very closely related to the `Collector` interface - more closely than the remaining static method in class `Collectors`. These two static interface methods illustrate what might become a Java programming idiom for builders or factories.

The static methods in interface `Collector` are two overloaded versions of a factory method named `of`. As arguments the factory methods receive all the parts that a collector consists of, create a new collector from these parts, and return it.

A collector consists of 5 parts:

- a *supplier* function that creates a new result container,
- an *accumulator* function that incorporates a new data element into a result container,
- a *combiner* function that combines two result containers into one,
- a optional *finisher* function that performs a final transform on the result container, and
- the collector *characteristics* that provide hints for an implementation with better performance.

The `Collector` interface has an abstract getter method per part and a factory method that takes an argument per part:

```
public interface Collector<T, A, R> {
  Supplier<A>          supplier();
  BiConsumer<A, T>     accumulator();
  BinaryOperator<A>    combiner();
  Function<A, R>       finisher();
  Set<Characteristics> characteristics();
  …
  public static<T,A,R> Collector<T,A,R> of(
    Supplier<A>        supplier,
    BiConsumer<A, T>   accumulator,
    BinaryOperator<A>  combiner,
    Function<A, R>     finisher,
    Characteristics... characteristics) {
      return new CollectorImpl<>(supplier,
                                 accumulator,
                                 combiner,
                                 finisher,
```

```
                                    characteristics);
    }
}
```

There is an additional second version of the factory method with only four arguments: it omits the optional finisher part.

The implementing class `CollectorImpl` is also closely related. It has five instance fields and a constructor with five arguments:

```
class CollectorImpl<T, A, R> implements Collector<T, A, R> {
        private final Supplier<A> supplier;
        private final BiConsumer<A, T> accumulator;
        private final BinaryOperator<A> combiner;
        private final Function<A, R> finisher;
        private final Set<Characteristics> characteristics;

        CollectorImpl(Supplier<A> supplier,
                      BiConsumer<A, T> accumulator,
                      BinaryOperator<A> combiner,
                      Function<A,R> finisher,
                      Set<Characteristics> characteristics) {
            this.supplier = supplier;
            this.accumulator = accumulator;
            this.combiner = combiner;
            this.finisher = finisher;
            this.characteristics = characteristics;
        }
        ...
        public BiConsumer<A, T> accumulator() {
            return accumulator;
        }
        public Supplier<A> supplier() {
            return supplier;
        }
        public BinaryOperator<A> combiner() {
            return combiner;
        }
        public Function<A, R> finisher() {
            return finisher;
        }
        public Set<Characteristics> characteristics() {
            return characteristics;
        }
    }
```

There is a second constructor with only four arguments; like the second factory method it omits the optional finisher part. The implementing `CollectorImpl` class is defined as a nested class in the companion class `Collectors`.

The two static factory methods in interface `Collector` directly reflect the structure of a collector: a collector consists of 5 parts, the interface has 5 matching getter methods, the static factory method takes 5 corresponding

arguments, the implementing class has 5 fields and a constructor with 5 matching arguments.

The distinguishing feature of the two static factory methods in interface `Collector` becomes obvious when we compare them to the remaining ~30 static methods in the companion class `Collectors`. Examples of the static methods in class `Collectors are:`

```
public static <T> Collector<T,?,List<T>> toList()
public static <T> Collector<T,?,Set<T>> toSet()
```

They are factory methods, too, but there is no immediate connection to the structure of a collector.

Conclusion

As you tell from the examples it is largely a matter of preferences and style whether a static method is located in an interface or a class. In principle you can completely eliminate companion classes such as `Collections` or `Collectors`. Equally well you can ignore the new feature of static interface methods altogether. What will become common practice remains to be seen. Or, as Brian Goetz put it: " *So, while this gives API designers one more tool, there don't seem to be obvious hard and fast rules about how to use this tool yet, and the simple-minded "all or nothing" candidates are likely to give the wrong result.* " [11]

[11] See http://mail.openjdk.java.net/pipermail/lambda-dev/2013-April/009345.html

Programming with Lambdas

Lambda expressions and method/constructor references were added to the Java programming language in order to ease the use of the stream API. Naturally, they can be used independently of streams. In the following section we want to explore the use of lambdas in general.

Programming with lambda expressions and method/constructor references has two aspects: usage and design.

Usage. You can use lambdas for ad-hoc definition of functionality. This is what they are for. Typically you will pass these lambdas to operations that take functions. The JDK's collection framework and in particular its stream abstraction in package `java.util.stream` is an example: it has an abundance of operations that take functions as arguments. These operations would be hard to use without lambda expressions and method/constructor references. When you use lambdas you need to learn the syntax of lambda expressions and method/constructor references and, of course, the API to which you intend to supply the lambdas.

Design. An entirely different aspect is the design of functional APIs. When you design an API that others will be using you decide in which way it will be used. Traditional APIs in Java were object-oriented in nature and demanded an imperative programming style. In the future you will have the option to design a more functional API in Java that encourages a functional programming style. The JDK's stream API is an example, but there are more opportunities for useful functional APIs.

Using lambdas is easy. Once you have read the *Lambda Tutorial* and some or all of this *Lambda Reference* you will know lambdas well enough to use them in conjunction with functional APIs such as the JDK's stream API. One piece that is still missing is to familiarize you with the JDK stream. If you want to learn more about the stream API read the *Stream Tutorial* and *the Stream Reference*.

In the following sections we want to take a closer look at the design of functional APIs - and related complications such as checked exceptions and generics. We will re-visit the Execute-Around-Method pattern mentioned in the *Lambda Tutorial* as the main example of a functional programming idiom.

The Execute-Around-Method Pattern

In the *Lambda Tutorial* we mentioned the Execute-Around-Method pattern as a programming technique for eliminating code duplication.

The Execute-Around-Method pattern addresses situations where it is required that some boilerplate code must be executed both before and after a method (or more generally, before and after another piece of code that varies). Often we simply duplicate the boilerplate code via copy-and-paste and insert the variable functionality manually. Following the DRY (don't repeat yourself) principle you might want to remove the code duplication via copy-and-paste. For this purpose it is necessary to separate the boilerplate code from the variable code. The boilerplate code can be expressed as a method and the variable piece of code can be passed to this method as the method's argument. This is an idiom where functionality (i.e. the variable piece of code) is passed to a method (i.e. the boilerplate code). The functionality can be conveniently and concisely be expressed by means of lambda expressions.

An example is the use of explicit locks. An explicit `ReentrantLock` (from package `java.util.lock`) must be acquired and released before and after a critical region of statements. Hence the boilerplate code looks like this:

```
class SomeClass {
  private ... some data ...
  private Lock lock = new ReentrantLock();
  ...
  public void someMethod() {
    lock.lock();
    try {
       ... critical region ...
    } finally {
       lock.unlock();
    }
  }
}
```

In all places where we need to acquire and release the lock the same boilerplate code of "lock-try-finally-unlock" appears. Following the Execute-Around-Method pattern we factor out the boilerplate code into a helper method:

```
class Utilities {
  public static void withLock(Lock lock, CriticalRegion cr) {
    lock.lock();
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
```

```
    }
  }
```

The helper method `withLock` takes the variable code as a method argument of type `CriticalRegion`:

```
@FunctionalInterface
public interface CriticalRegion {
  void apply();
}
```

The interface `CriticalRegion` is a functional interface and hence a lambda expression can be used to provide an implementation of the `CriticalRegion` interface.

Now we want to use the `withLock` utility to get rid of code duplication in the implementation of a `Stack` class. Here is a piece of code from a `Stack` class's implementation that uses the helper method `withLock`:

```
private class Stack<T> {
  private Lock lock = new ReentrantLock();
  @SuppressWarnings("unchecked")
  private T[] array = (T[])new Object[16];
  private int sp = -1;

  public void push(T e) {
    withLock(lock, () -> {
      if (++sp >= array.length)
        resize();
      array[sp] = e;
    });
  }

  ...
}
```

The boilerplate code is reduced to invocation of the `withLock` helper method and the critical region is provided as a lambda expression. While the suggested `withLock` method indeed aids elimination of code duplication it is by no means sufficient as a multi-purpose utility. There are several open issues. What if ...

- the critical region needs access to data from its enclosing context, perhaps even mutating access, or
- the critical region returns a value, or
- the critical region throws exceptions, perhaps even checked exceptions.

Data Access

Access to the enclosing scope's data is usually not a problem. Lambdas can have bindings to outer scope variables as long as the variables are effectively final. Plus, lambdas have unrestricted access to the enclosing class's fields. The code snippet above already demonstrates the data access.

```
private class Stack<T> {
  private Lock lock = new ReentrantLock();
  @SuppressWarnings("unchecked")
  private T[] array = (T[])new Object[16];
  private int sp = -1;

  public void push(T e) {
    withLock(lock, () -> {
      if (++sp >= array.length)
        resize();
      array[sp] = e;
    } );
  }

  ...
}
```

The lambda expression reads the enclosing `push` method's argument `e`. It also modifies the enclosing `Stack` class's fields `sp` and `array`.

Return Value

The return value is a little more difficult to handle. For instance, the `Stack` class's `pop` method has a return value, but our `withLock` utility does not accept critical regions that return a value. In order to allow for lambda expressions with a return type different from `void` we can use an additional `CriticalRegion` interface with an `apply` method that returns a result. This way we end up with two interfaces:

```
@FunctionalInterface
public interface VoidCriticalRegion {
  void apply();
}
@FunctionalInterface
public interface GenericCriticalRegion<R> {
  R apply();
}
```

Inevitably, we also need an additional helper method.

```
class Utilities {
  public static
  void withLock(Lock lock,VoidCriticalRegion cr) {
    lock.lock();
```

```
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
  }
  public static <R> R withLock
  (Lock lock, GenericCriticalRegion<? extends R> cr) {
    lock.lock();
    try {
      return cr.apply();
    } finally {
      lock.unlock();
    }
  }
}
```

Given the additional helper method and functional interface the `pop` method can be implemented like this:

```
private class Stack<T> {

  ...

  public T pop() {
   return withLock(lock, () -> {
      if (sp < 0)
        throw new NoSuchElementException();
      else
        return array[sp--];
     });
  }
}
```

Note, that we now have two functional interfaces for the critical region and two `withLock` utility methods: one for critical regions with a reference return type and one for a void return type. The additional interface and method change the API, but they do not affect its usage. The user simply passes adequate functionality to the `withLock` utility method and the compiler's overload resolution mechanism determines which of the utility methods must be invoked.

Primitive Types

A general purpose `withLock` utility might need further variants for each of the primitive types as return types i.e., an `IntCriticalRegion`, a `LongCriticalRegion`, a `DoubleCriticalRegion`, etc. Naturally, we need corresponding `withLock` helper methods, so that we end up with many more functional interfaces and helper methods. Unfortunately, this drastically increases the opportunity for ambiguities.

Here is an example:

```
@FunctionalInterface
public interface VoidCriticalRegion {
  void apply();
}
@FunctionalInterface
public interface GenericCriticalRegion<R> {
  R apply();
}
@FunctionalInterface
public interface IntCriticalRegion {
  int apply();
}
```

Inevitably, we also need additional helper methods.

```
class Utilities {
  public static
  void withLock(Lock lock,VoidCriticalRegion cr) {
    lock.lock();
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
  }
  public static
  <R> R withLock
  (Lock lock, GenericCriticalRegion<? extends R> cr) {
    lock.lock();
    try {
      return cr.apply();
    } finally {
      lock.unlock();
    }
  }
  public static
  int withLock(Lock lock, IntCriticalRegion cr) {
    lock.lock();
    try {
      return cr.apply();
    } finally {
      lock.unlock();
    }
  }
}
```

A class for a stack of primitive type `int` values would look like this:

```
public class IntStack {
        private Lock lock = new ReentrantLock();
        private int[] array = new int[16];
        private int sp = -1;
        ...
        public void push(int e) {
```

```
            withLock(lock, () -> {
                if (++sp >= array.length)
                    resize();
                array[sp] = e;
            });
        }
        public int pop() {
            return withLock(lock, (IntCriticalRegion) () -> {
                if (sp < 0)
                    throw new NoSuchElementException();
                else
                    return (array[sp--]);
            });
        }
}
```

Note the ugly cast in the `pop` method. It is needed because the compiler yields an error message without it; it cannot figure out whether we are asking for the `withLock` version with a `GenericCriticalRegion<Integer>` and or with a `IntCriticalRegion`. Hence the cast is mandatory - which almost defeats the purpose of using the `withLock` utility in the first place.

We can get rid of the cast by discarding the primitive type versions of the `CriticalRegion` interface and the related overloads of the `withLock` method. It reduces the number of overloaded methods and thereby the chance for ambiguity. In return, we would have to accept the overhead of boxing and unboxing, and can hope that the compiler's optimization strategy might eliminate the overhead.

The point to take home is that additional functional interfaces for the primitive types might look attractive because they eliminate the boxing/unboxing overhead. At the same time they increase the risk of overload resolution failure at compile-time and might not be worth the trouble they cause.

Unchecked Exceptions

Critical regions that throw unchecked exception do not need any particular attention because unchecked exceptions need not be listed in throws clauses, neither in the functional interface nor in the utility method.

The `pop` method for instance throws an unchecked exception:

```
private class Stack<T> {

   ...

   public T pop() {
    return withLock(lock, () -> {
       if (sp < 0)
```

```
          throw new NoSuchElementException();
        else
          return array[sp--];
      });
    }
  }
```

This works although neither the critical region's `apply` method nor the `withLock` utility method have a `throws` clause.

Checked Exceptions

The situation is different if the critical region throws *checked* exceptions. For illustration let us modify the `pop` method so that it throws a checked exception if the stack is empty.

```
public class Stack<T> {
  …
  public static class EmptyStackException extends Exception {}

  public T pop() throws EmptyStackException {
    return withLock(lock, () -> {
      if (sp < 0)
        throw new EmptyStackException();
      else
        return (array[sp--]);
    });
  }
}
```

How do we cope with critical regions that do throw checked exceptions? So far, this does not compile because neither the critical region's `apply` method nor the `withLock` utility method are allowed to raise checked exceptions.

In principle there are several strategies for solving the problem:

- *Exception Tunnelling*. We wrap all checked exceptions into unchecked exception and unwrap them later. This way we need not add throws clauses the functional interface or the utility method.
- *Adding Throws Clauses*. We generify the the functional interface or the utility method by adding a type variable for the exception type.

Exception Tunnelling

One technique is wrapping all checked exceptions into unchecked exception and unwrapping them later.

Here is an example:

```
public T pop() throws EmptyStackException {
```

```
  try {
    return withLock(lock, () -> {
      try {
        if (sp < 0)
          throw new EmptyStackException();
        else
          return (array[sp--]);
      } catch (Exception e) {
         throw new RuntimeException(e);
      }
    });
  } catch (final RuntimeException re) {
    Throwable cause = re.getCause();
    if (cause instanceof EmptyStackException)
      throw ((EmptyStackException) cause);
    else
      throw re;
  }
}
```

The critical region lambda throws a checked `EmptyStackException`, which the functional interface's `apply` method does not permit. As a work-around the `EmptyStackException` is wrapped in to a runtime exception. The downside of this *tunnelling* technique is that the receiving code must catch the runtime exception, unwrap it and thereby restore the original checked exception.

Adding Throws Clauses

An alternative would involve additional helper methods and functional interfaces that have appropriate throws clauses. In order to avoid the exception tunnelling for the `pop` method from the example above we need to change the critical region interface and the `withLock` utility as follows:

```
@FunctionalInterface
public interface CriticalRegion<R> {
    R apply() throws EmptyStackException;
}

public static
<R> R withLock(Lock lock, GenericCriticalRegion<? extends R> cr)
throws EmptyStackException {
  lock.lock();
  try {
    return cr.apply();
  } finally {
    lock.unlock();
  }
}
```

This simplifies the user code substantially because there is no need for exception wrapping and unwrapping anymore:

```
public T pop() throws EmptyStackException {
  return withLock(lock, () -> {
    if (sp < 0)
      throw new EmptyStackException();
    else
      return (array[sp--]);
  });
}
```

The downside is that we need one additional pair of helper method and functional interface per variation of the throws clause. The solution strategy demonstrated above can be generalized by means of generics: we can add a type parameter for the exception type.

With a type parameter for the exception type the helper method and functional interface look like this:

```
@FunctionalInterface
public interface CriticalRegion<R, E extends Exception> {
    R apply() throws E;
}

public static <R, E extends Exception> R withLock
  (Lock lock, GenericCriticalRegion<? extends R, ? extends E> cr)
  throws E {
    lock.lock();
    try {
      return cr.apply();
    } finally {
      lock.unlock();
    }
}
```

The user code, e.g. the stack's `pop` method, is not affected at all and we can now use the generic `withLock` utility method for all critical regions with a checked exception regardless of the exact exception type.

Unfortunately, the generic throws clause does not help if the critical region throws *more than one exception type*. For illustration we change the `push` method so that it throws two types of exceptions:

```
public class Stack<T> {
  …
  public static class SizeLimitExceededException
    extends Exception {}
  public static class IllegalElementException
    extends Exception {}

  public void push(T e)                                    // error
    throws SizeLimitExceededException, IllegalElementException {
      withLock(lock, () -> {
        if (++sp >= array.length)
          throw new SizeLimitExceededException();
        if (e == null)
```

```
            throw new IllegalElementException();
          array[sp] = e;
      });
    }
  }
```

The compiler now complains about an unreported exception of type `Exception` raised by the lambda expression. This is because the compiler deduces that the lambda expression throws an `Exception` (which is the common supertype of two actually raised exceptions `SizeLimitExceededException` and `IllegalElementException`). For this reason the `push` method is required to take care of the `Exception` by either catching or declaring it in its throws clause. Neither is desirable; we do not want to change the `push` method.

As a solution we could consider defining yet another pair of functional interface and utility method with two type variables for two different exception types (and so on and so forth for situations with 3, 4, 5, or more exception types). The unfortunate part is that we cannot even overload on exception clauses of different length.

Let us try to provide the additional functional interface in order to illustrate the dilemma:

```
@FunctionalInterface
public interface VoidCriticalRegion<E extends Exception> {
  void apply() throws E;
}
@FunctionalInterface                                        // error
public interface VoidCriticalRegion<E1 extends Exception,
                                    E2 extends Exception> {
  void apply() throws E1, E2;
}
```

The compiler immediately complains that the two interfaces have the same erasure and rejects our attempt to provide a second functional interface with the same name. Naturally, we can choose a different name. e.g. `VoidCriticalRegion_2`. We also need to provide a corresponding utility method:

```
@FunctionalInterface
public interface VoidCriticalRegion<E extends Exception> {
  void apply() throws E;
}
@FunctionalInterface
public interface VoidCriticalRegion_2<E1 extends Exception,
                                      E2 extends Exception> {
  void apply() throws E1, E2;
}

public static <E extends Exception>
void withLock
```

```
(Lock lock, VoidCriticalRegion<? extends E> cr)
throws E {
    lock.lock();
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
}
public static <E1 extends Exception, E2 extends Exception>
void withLock
(Lock lock, VoidCriticalRegion_2<? extends E1, ? extends E2> cr)
throws E1, E2 {
    lock.lock();
    try {
      cr.apply();
    } finally {
      lock.unlock();
    }
}
```

So far, it compiles. But, when we invoke the overloaded `withLock` method then the compiler runs into overload resolution problems and reports ambiguities. Here is what happens:

```
public void push(T e)
throws SizeLimitExceededException, IllegalElementException {
  withLock(lock, () -> {                                    // error
    if (++sp >= array.length)
          throw new SizeLimitExceededException();
    if (e == null)
          throw new IllegalElementException();
    array[sp] = e;
  });
}
```

The compiler finds both `withLock` methods, considers both viable, and reports an ambuity. We can now rename the second withLock method to withLock_2 (or `withLockForCriticalRegionsWithTwoExceptionTypes`). Whatever we try, ultimately coping with checked exceptions in design of functional APIs is ugly and quite a mess.

Exception Transparency

To address this mess, a compiler strategy called *exception transparency* was discussed. Exception transparency means that the compiler automatically infers the throws clauses of methods like `withLock` in our example.

This kind of compiler support would eliminate the need for countless variations of the helper method and the functional interface. For the time

being the language designers decided against exception transparency. The feature may still be added to the language is a future release of Java.[12]

For Java 8 it means that tunnelling is the technique of choice in order to cope with checked exceptions.

Checked Exception in Functional APIs of the JDK

Note, that the complications with checked exception are not limited to our example. The JDK streams and their bulk operation struggle with the same issue. All the functional interfaces that are used in conjunction with `forEach`, `filter`, `map`, `reduce`, etc. do not allow checked exceptions. In practice, wrapping checked exceptions into runtime exceptions is the norm.

The `lines` method in class `java.io.BufferedReader` illustrates this. It returns a `Stream<String>` polulated with the lines read from a `BufferedReader`. Accessing the underlying `BufferedReader` may cause a checked `IOException`. As stream operations cannot handle checked exceptions the checked `IOException` is wrapped in an `UncheckedIOException`, whose sole purpose is tunnelling i/o related checked exceptions.

Wildcards Instantiations of Functional Interfaces

Many functional interfaces are generic. In the example discussed in the previous sections we ended up with a generic functional interface for the critical region. The same can be observed in the JDK. Just take a look at the `java.util.function` package from the JDK: all functional interfaces in this package are generic interfaces.

Functional interface types are typically used as argument types of methods. In our example we passed the generic `CriticalRegion` interface as an argument to the `withLock` utility method. Similarly, the functional interfaces such as `Function`, `Consumer`, `Supplier`, etc. from the JDK package `java.util.function` are passed to various methods, e.g. to the stream operations defined in interface `Stream`.

It turns out that almost always the argument type must be a wildcard instantiation of the generic functional interface. As wildcards are an

[12] If you are interested in the considerations regarding exception transparency, here are a couple of references:
https://blogs.oracle.com/briangoetz/entry/exception_transparency_in_java and
http://mail.openjdk.java.net/pipermail/lambda-spec-experts/2012-September/000007.html.

aspect of generics that is considered "difficult" by many Java developers we take a closer look at the use of functional interfaces and the need for wildcard instantiations thereof.

To demonstrate and explain the need for wildcard instantiation we use a simple example: a `Filter` interface that is passed to the `filter` method of a `Sequence`.

Here is the functional interface `Filter`:

```
@FunctionalInterface
public interface Filter<T> {
   boolean isGood(T t);
}
```

This is the `Sequence` abstraction with its `filter` operation (1st approach, not yet perfect):

```
public class Sequence<T> {
    private List<T> seq = new ArrayList<>();
    private Sequence(List<T> source) {
        seq = source;
    }
    @SafeVarargs
    public Sequence(T... elems){
        seq = Arrays.asList(elems);
    }
    public Sequence<T> filter(Filter<T> filter) {
        List<T> res = new ArrayList<>();
        for (T t : seq)
            if (filter.isGood(t))
                res.add(t);
        return new Sequence<T>(res);
    }
    public String toString() {
        return seq.toString();
    }
}
```

Here we invoke the `filter` operation to which we pass the method reference `Character::isAlphabetic` as a filter function:

```
Sequence<Character> cs = new Sequence<>
          ('€','§','4','b','ß','Z','ö',(char)0x007E,(char)0x221E,
          (char)0x042F,(char)0x2167,(char) 0x2211);
System.out.println(cs);
Sequence<Character> rs = cs.filter(Character::isAlphabetic);
System.out.println(rs);
```

It prints:

```
[€, §, 4, b, ß, Z, ö, ~, ∞, Я, Ⅷ, ∑]
[b, ß, Z, ö, Я, Ⅷ]
```

All is fine so far. If, however, we use the `Filter` interface in a slightly different context it does no longer work:

```
public static
<E> Sequence<E> universalFilter(Sequence<E> s, Filter<Object> f){
  return s.filter(f);            // error: incompatible types
}

System.out.println(universalFilter(cs,o -> o.hashCode()%2!=0));
```

In the `universalFilter` method we intend to use a filter of type `Filter<Object>` that can handle any kind of object. When we invoke the method we pass in such a filter, namely `o->o.hashCode()%2!=0`. This filter indeed works for all reference types. One would expect that it should also work for the elements of unknown type `E` in the `Sequence<E>` that we pass to the `universalFilter` method along with the `Filter<Object>`.

Yet the compiler complains. It reports that `Filter<Object>` cannot be converted to `Filter<E>`, which is correct. `E` might be a type different from `Object`, which means that the two filter types `Filter<E>` and `Filter<Object>` are indeed incompatible types.

The problem is that the `filter` method in class `Sequence<T>` requires an argument of type `Filter<T>`, which is too restrictive. It prohibits the use of the perfectly reasonable universal filter of type `Filter<Object>`.

The correct signature of the `filter` method must look like this (2nd approach, much better):

```
public class Sequence<T> {
    ...
    public Sequence<T> filter(Filter<? super T> filter) {
        List<T> res = new ArrayList<>();
        for (T t : seq)
            if (filter.isGood(t))
                res.add(t);
        return new Sequence<T>(res);
    }
    ...
}
```

The `filter` method must allow parameterizations of the `Filter` interface for supertype of the sequence's element type `T` rather than demanding a filter of type `Filter<T>`. With this correction the sample code compiles and runs and prints:

```
[S, ß, Я, Ⅷ, ∑]
```

Here is an additional example. Say, we have a functional interface `Mapper`:

```
@FunctionalInterface
```

```
public interface Mapper<F,T> {
    T mapTo(F from);
}
```

It is used by the `Sequence`'s `map` operation (1st approach, not yet perfect):

```
public class Sequence<T> {
    private List<T> seq = new ArrayList<>();
    private Sequence(List<T> source) {
        seq = source;
    }
    …
    public <X> Sequence<X> map(Mapper<T,X> mapper) {
        List<X> buf = new ArrayList<>();
        for (T t : seq)
            buf.add(mapper.mapTo(t));
        return new Sequence<>(buf);
    }
}
```

We can use the `map` operation like this:

```
Sequence<Integer> is = new Sequence<>(10,20,30);
System.out.println(is);
Sequence<Double> ds = is.map(i->i/2.0);
System.out.println(ds);
```

It compiles and runs and prints:

```
[10, 20, 30]
[5.0, 10.0, 15.0]
```

Now imagine we have a universal mapper that can map any type of object to a character:

```
Mapper<Object,Character> mapper = o -> o.toString().charAt(0);
```

We want to use this mapper to map a sequence of integers to a sequence of characters like this:

```
Sequence<Character> os = is.map(mapper);  // error: type mismatch
```

The compiler complains because the `map` operation of a `Sequence<Integer>` is declared to take a mapper of type `Mapper<Integer,…>`, i.e. a mapper that take integers and maps them to something. We want to use a mapper of type `Mapper<Object,…>` that takes any kind of object (and in particular integers) and maps them to something. Our more universal mapper is perfectly reasonalbe; it is the `map` method that is too restrictive.

The signature of the `map` method can be relaxed like this (2nd approach, somewhat improved):

```
public class Sequence<T> {
```

```
    …
    public <X> Sequence<X> map(Mapper<? super T,X> mapper) {
        List<X> buf = new ArrayList<>();
        for (T t : seq)
            buf.add(mapper.mapTo(t));
        return new Sequence<>(buf);
    }
}
```

Our attempt to map a sequence of integers to a sequence of characters with a mapper of type `Mapper<Object,Character>` now compiles and runs:

```
Mapper<Object,Character> mapper = o -> o.toString().charAt(0);
Sequence<Character> os = is.map(mapper);              // now fine
```

If we want to use the same mapper to map a sequence of integers to a sequence of objects the compiler complains again:

```
Mapper<Object,Character> mapper = o -> o.toString().charAt(0);
Sequence<Object> os = is.map(mapper);     // error: type mismatch
```

This time it is return type that causes the trouble. If the `map` operation receives a mapper of type `Mapper<…,Character>` then it returns a `Sequence<Character>`. We, instead, expect a `Sequence<Object>`. Obviously, a sequence of objects can store the results of any mapping and in particular the characters that are produced by our mapper of type `Mapper<…, Character>`. Again, it is the `map` method that is too restrictive.

Here is the final correction of the `map` method (3rd approach, maximally relaxed):

```
public class Sequence<T> {
    …
    public <X> Sequence<X>
    map(Mapper<? super T,? extends X> mapper) {
        List<X> buf = new ArrayList<>();
        for (T t : seq)
            buf.add(mapper.mapTo(t));
        return new Sequence<>(buf);
    }
}
```

Our attempt to store the result of the mapping to a sequence of objects now compiles and runs:

```
Mapper<Object,Character> mapper = o -> o.toString().charAt(0);
Sequence<Object> os = is.map(mapper);                 // now fine
```

The admittedly contrived example demonstrates a common situation in functional API design. Many of the functional interfaces are generic interfaces. The operations that take implementations of these generic

functional interfaces must often declare wildcard parameterizations of the functional interfaces as their argument types in order to be correct. [13]

[13] For more information on wildcards see
http://www.angelikalanger.com/GenericsFAQ/FAQSections/Index.html#W

Runtime Representation of Lambda Expressions

In this section we want to explore what lambda expression and method/constructor are translated to and how they are serialized.

Translation of Lambda Expressions

to be done

Serialization of Lambda Expressions

to be done

Appendix

Source Code of Execute-Around-Method Pattern Case Study

withLock Utility

```
public class Utilities {
   @FunctionalInterface
   public static interface VoidCriticalRegion {
        void apply();
   }
   public static
   void withLock(Lock lock, VoidCriticalRegion region) {
        lock.lock();
        try {
            region.apply();
        } finally {
            lock.unlock();
        }
   }
   @FunctionalInterface
   public static interface GenericCriticalRegion<R> {
        R apply();
   }
   public static
   <R> R withLock(Lock lock, GenericCriticalRegion<R> region) {
        lock.lock();
        try {
            return region.apply();
        } finally {
            lock.unlock();
        }
   }
}
```

Stack Class

```
public class Stack<T> {
        private Lock lock = new ReentrantLock();
        @SuppressWarnings("unchecked")
        private T[] array = (T[])new Object[16];
        private int sp = -1;

        private void resize() {
            // todo later - for now throw index out of bounds
            array[sp] = null;
```

```
        }

        public void push(T e) {
            withLock(lock, () -> {
                if (++sp >= array.length)
                    resize();

                array[sp] = e;
            });
        }

        public T pop() {
            return withLock(lock, () -> {
                if (sp < 0)
                    throw new NoSuchElementException();
                else
                    return (array[sp--]);
            });
        }
}
```

Experiments with IOException

```
public class IOSample {
  private Lock lock = new ReentrantLock();
  private IntStack stack = new IntStack();

  /*
   *  This uses the VoidCriticalRegion functional interface.
   */
  public void myMethod_1() throws IOException {
    try {
      withLock(lock, () -> {
        try {
          InputStream is = new FileInputStream("test");
          if (is.available() <= 0)
            stack.push(Integer.MAX_VALUE);
          else
            stack.push(Integer.MIN_VALUE);
          } catch (IOException ioe) {
            throw new RuntimeException(ioe);
          }
      });
    } catch (RuntimeException re) {
      Throwable cause = re.getCause();
      if (cause instanceof IOException)
        throw ((IOException) cause);
      else
        throw re;
    }
  }
  /*
   *  This uses the VoidIOECriticalRegion functional interface.
   */
```

```
    public void myMethod_2() throws IOException {
      withLockAndIOE(lock, () -> {
        InputStream is = new FileInputStream("test");
        if (is.available() <= 0)
          stack.push(Integer.MAX_VALUE);
        else
          stack.push(Integer.MIN_VALUE);
      });
    }
  }
```

Index