Variable arity constructors and methods

Rule

The last parameter of a constructor or method can have variable arity

Example of declaration

```
public class VariableArity {
    public static int max(int first, int... others) {
        // in the body 'others' is considered an array of type 'int[]'
        int res = first;
        for (int e : others)
            if (e > res)
                res = e;
        return res;
    }
}
```

`max()` can be called with

- one or more arguments of type **`int`**
- a first argument of type **`int`**, and a second one of type **`int`**`[]`

Variable arity constructors and methods

Demo

```
public class VariableArity {
    public static int max(int first, int... others) {
        // 'others' treated as an array of type 'int[]'
        int res = first;
        for (int e : others)
            if (e > res)
                res = e;
        return res;
    }

    public static void main(String[] args) {
        assert VariableArity.max(42) == 42;
        assert VariableArity.max(42, 50) == 50;
        assert VariableArity.max(42, 50, 49) == 50;
        assert VariableArity.max(42, new int[]{50, 49, 52}) == 52;
    }
}
```

Constructor and method overloading

Constructor and method signature

- constructors are distinguished by their signature: the number and types of parameters
- methods are distinguished by their signature: the name and the number and types of parameters
- both object and class methods can be overloaded
- not allowed to declare constructors/methods with the same signature

Remarks: difference between members and declared members

- Constructors of class *C*
  - are only declared in *C*, they cannot be inherited
- Methods of class *C* are those
  - declared in *C* and in any implemented interface
  - inherited from superclasses
- Recall: private methods cannot be inherited
- Remark: since Java 8 (2014) class methods are allowed in interfaces

Constructor and method overloading

Problem

Calls to overloaded constructors or methods may be ambiguous

Demo with `java.io.PrintStream`

```
public class System {
    public static final PrintStream out;
    ...
}
public class PrintStream extends FilterOutputStream ... {
    public void println(){...}
    public void println(char[] s){...}
    public void println(Object obj){...}
    public void println(String s){...}
    ... // more other 'println' methods
}
public class PrintTest {
    public static void main(String[] args) {
        char[] a = {'t', 'e', 's', 't'};
        Object o = a;
        System.out.println(a); // which method is called?
        System.out.println(o); // which method is called?
    }
}
```

Constructor and method overloading

Problem

Calls to overloaded constructors or methods may be ambiguous

Demo with `java.io.PrintStream`

```
public class System {
    public static final PrintStream out;
    ...
}
public class PrintStream extends FilterOutputStream ... {
    public void println(){...}
    public void println(char[] s){...}
    public void println(Object obj){...}
    public void println(String s){...}
    ... // more other 'println' methods
}
public class PrintTest {
    public static void main(String[] args) {
        char[] a = {'t', 'e', 's', 't'};
        Object o = a;
        System.out.println(a); // 'println(char[])' is called
        System.out.println(o); // 'println(Object)' is called
    }
}
```

Constructor and method overloading

Definition

Overloading resolution of a constructor or method call:

the selection of the signature of the constructor or method to be called

General rule

Overloading resolution in Java is static:

the signature is selected at compile-time and depends on the static types of the target and of the arguments

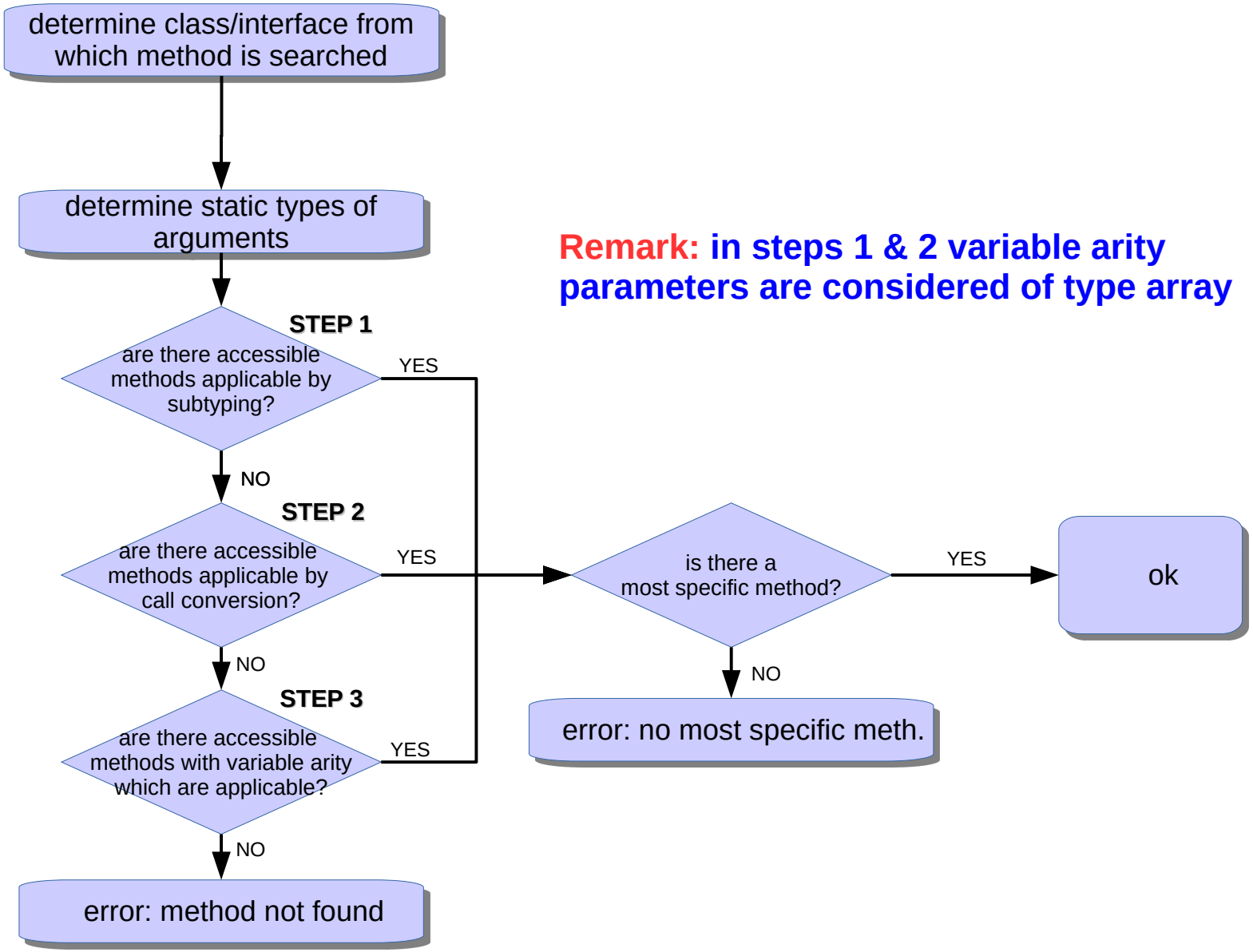
Remark: in constructors and class methods no target is involved, but resolution depends on the class where the code is defined

Example

```
char[] a = {'t','e','s','t'};
Object o = a;

System.out.println(a); // 'a' has static type char[] and dynamic type char[]
System.out.println(o); // 'o' has static type Object and dynamic type char[]
```

Detailed rules for overloading resolution



Remark: in steps 1 & 2 variable arity parameters are considered of type array

Step 1: applicability by subtyping

Example

```
public class Overload {
    public String m(int i) { return "int"; }                    // int ≤ int
    public String m(float f) { return "float"; }                // int ≤ float
    public String m(Number n) { return "Number"; }              // int ≰ Number
    public String m(Number... nums) { return "Number..."; } // int ≰ Number[]
    public static void main(String[] args) {
        Overload o=new Overload();
        assert o.m(42).equals("int");
    }
}
```

Resolution process for `o.m(42)`

1. the method is searched starting from the static type `Overload` of `o`
2. the static type of the argument is **`int`**
3. accessible methods applicable by subtyping: `m(int)` and `m(float)`
4. `m(int)` is the most specific signature, since **`int`** $\leq$ **`float`**
5. `o.m(42)` resolved with `m(int)`

More specific signature

Definition

$m(S_1, \ldots, S_n)$ is more specific than $m(T_1, \ldots, T_n)$

if and only if

$S_1 \leq T_1, \ldots, S_n \leq T_n$

Examples of more specific methods

- `println(char[])` more specific than `println(Object)`
- `println(String)` more specific than `println(Object)`
- `println(int)` more specific than `println(long)`
- `println(long)` more specific than `println(float)`
- `println(float)` more specific than `println(double)`

More specific signature

Definition

$m(S_1, \ldots, S_n)$ is more specific than $m(T_1, \ldots, T_n)$

if and only if

$S_1 \leq T_1, \ldots, S_n \leq T_n$

Examples of methods where none is more specific

- none more specific between `println(char[])` and `println(String)` because `char[]` $\not\leq$ `String` and `String` $\not\leq$ `char[]`
- none more specific between `replace(char,char)` and `replace(CharSequence,CharSequence)` because `char` $\not\leq$ `CharSequence` and `CharSequence` $\not\leq$ `char`
- none more specific between `lastIndexOf(int,int)` and `lastIndexOf(String,int)` because `int` $\not\leq$ `String` and `String` $\not\leq$ `int`

Step 2: applicability by call conversion

Example

```
public class Overload {
    public String m(int i) { return "int"; }                    // double ⋢ int
    public String m(float f) { return "float"; }                // double ⋢ float
    public String m(Number n) { return "Number"; }              // double ⋢ Number
    public String m(Number... nums) { return "Number..."; } // double ⋢ Number[]
    public static void main(String[] args) {
        Overload o=new Overload();
        assert o.m(42.0).equals("Number");
    }
}
```

Resolution process for `o.m(42.0)`

1. the method is searched starting from the static type `Overload` of `o`
2. the static type of the argument `42.0` is **`double`**
3. no accessible methods applicable by subtyping
4. accessible methods applicable by call conversion: `m(Number)` only
   `double` boxed to `Double`, `Double` $\leq$ `Number`, `Double` $\not\leq$ **`int`**, **`double`**, `Number[]`
5. `m(Number)` is the most specific signature, since it is the only one
6. `o.m(42.0)` resolved with `m(Number)`

Step 2: applicability by call conversion

Rule

The static types of the arguments can be converted to the static types of the parameters, if possible, in one of the following ways:

1. boxing + optional widening conversion between reference types
2. unboxing + optional widening conversion between primitive types

Examples

1. **`double`** → `Double` (boxing) `Double` → `Number` (widening)
2. `Float` → **`float`** (unboxing) **`float`** → **`double`** (widening)

Step 3: applicability with variable arity

Example

```
public class Overload {
    public String m(int i) { return "int"; }
    public String m(float f) { return "float"; }
    public String m(Number n) { return "Number"; }
    public String m(Number... nums) { return "Number..."; }
    public static void main(String[] args) {
        Overload o=new Overload();
        assert o.m(42,42.0).equals("Number...");
    }
}
```

Resolution process for `o.m(42,42.0)`

1. the method is searched starting from the static type `Overload` of `o`
2. the static types of the arguments are **`int`** and **`double`**
3. no accessible methods applicable by subtyping or call conversion with two parameters
4. `m(Number...)` is applicable by call conversion:
   `int`,**`double`** boxed to `Integer`,`Double`$\leq$`Number`
5. `m(Number...)` is the most specific signature, since it is the only one
6. `Overload.m(42,42.0)` resolved with `m(Number...)`

Syntactic abbreviations

Abbreviation for fields and methods

- object fields and methods

  this*.*f *and* ***this*.*m*** *(...) abbreviated with* ***f*** *and* ***m*** *(...)*

- class fields and methods

  C.*f* *and C*.*m* *(...) abbreviated with* ***f*** *and* ***m*** *(...)*

Example

```
public class Item {
    private static long availableSN;
    private int price;
    public final long serialNumber;
    public Item(int price) {
        if (price < 0)
            throw new IllegalArgumentException();
        this.price = price;
        this.serialNumber = Item.availableSN++;
    }
    public int getPrice() {
        return this.price;
    }
}
```

Syntactic abbreviations

Abbreviation for fields and methods

- object fields and methods

  this. *f and* ***this.*** *m (...) abbreviated with f and m (...)*

- class fields and methods

  C . f and C . m (...) abbreviated with f and m (...)

Example

```
public class Item {
    private static long availableSN;
    private int price;
    public final long serialNumber;
    public Item(int price) {
        if (price < 0)
            throw new IllegalArgumentException();
        this.price = price;             // cannot be abbreviated
        serialNumber = availableSN++; // abbreviated
    }
    public int getPrice() {
        return price;                   // abbreviated
    }
}
```

Syntactic abbreviations

Beware of ambiguities

- *f*: object field, class field, parameter or local variable?
- *m*(...): class or object method?

Example

```
public class Item {
    private static long availableSN;
    private int price;
    public final long serialNumber;
    public Item(int price) {
        if (price < 0)
            throw new IllegalArgumentException();
        this.price = price;              // cannot be abbreviated
        serialNumber = availableSN++; // abbreviated
    }
    public int getPrice() {
        return price;                    // abbreviated
    }
}
```