# Chapter5 Methods

Thursday, October 9, 2025 10:40 AM

## !!Attention

```
public class ParkTrip {
   public void skip1() {}
   default void skip2() {} // DOES NOT COMPILE
   void public skip3() {}  // DOES NOT COMPILE
   void skip4() {}
}
```

```
public class Mike {
   public void hike1() {}
   public void hike2() { return; }
   public String hike3() { return ""; }
   public String hike4() {}            // DOES NOT COMPILE
   public hike5() {}                   // DOES NOT COMPILE
   public String int hike6() { }       // DOES NOT COMPILE
   String hike7(int a) {               // DOES NOT COMPILE
      if (1 < 2) return "orange";
   }
}
```

```
public class Exercise {
   public void bike1() {}
   public final void bike2() {}
   public static final void bike3() {}
   public final static void bike4() {}
   public modifier void bike5() {}       // DOES NOT COMPILE
   public void final bike6() {}          // DOES NOT COMPILE
   final public void bike7() {}
}
```

```
public class BeachTrip {
   public void jog1() {}
   public void 2jog() {}   // DOES NOT COMPILE
   public jog3 void() {}   // DOES NOT COMPILE
   public void Jog_$() {}
   public _() {}           // DOES NOT COMPILE
   public void() {}        // DOES NOT COMPILE
}
```

### Accessing a Static Variable or Method

Usually, accessing a `static` member is easy.

```
public class Snake {
   public static long hiss = 2;
}
```

You just put the class name before the method or variable, and you are done. Here's an example:

```
System.out.println(Snake.hiss);
```

Nice and easy. There is one rule that is trickier. You can use an instance of the object to call a `static` method. The compiler checks for the type of the reference and uses that instead of the object—which is sneaky of Java. This code is perfectly legal:

```
5: Snake s = new Snake();
6: System.out.println(s.hiss);  // s is a Snake
7: s = null;
8: System.out.println(s.hiss);  // s is still a Snake
```

### Varargs

Which method do you think is called if we pass an `int[]`?

```
public class Toucan {
   public void fly(int[] lengths) {}
   public void fly(int… lengths) {}     // DOES NOT COMPILE
}
```

Trick question! Remember that Java treats varargs as if they were an array. This means the method signature is the same for both methods. Since we are not allowed to overload methods with the same parameter list, this code doesn't compile. Even though the code doesn't look the same, it compiles to the same parameter list.

## Method Name:

```
public class Measurement {
   int getHeight1() {
      int temp = 9;
      return temp;
   }
   int getHeight2() {
      int temp = 9L;  // DOES NOT COMPILE
      return temp;
   }
   int getHeight3() {
      long temp = 9L;
      return temp;    // DOES NOT COMPILE
   }
}
```

```
public class Bird {
   public void fly1() {}
   public void fly2()          // DOES NOT COMPILE
   public void fly3(int a) { int name = 5; }
}
```

The `fly1()` method is a valid declaration with an empty method body. The `fly2()` method doesn't compile because it is missing the braces around the empty method body. Methods are required to have a body unless they are declared `abstract`. We cover `abstract` methods in Chapter 6, "Class Design." The `fly3()` method is a valid declaration with one statement in the method body.

```
public void zooAnimalCheckup(boolean isWeekend) {
   final int rest;
   if(isWeekend) rest = 5;
   System.out.print(rest);  // DOES NOT COMPILE
}
```

Does using the `final` modifier mean we can't modify the data? Nope. The `final` attribute refers only to the variable reference; the contents can be freely modified (assuming the object isn't immutable).

The `rest` variable might not have been assigned a value, such as if `isWeekend` is `false`. Since the compiler does not allow the use of local variables that may not have been assigned a value, the code does not compile.

#### Effectively Final Variables

An *effectively final* local variable is one that is not modified after it is assigned. This means that the value of a variable doesn't change after it is set, regardless of whether it is explicitly marked as `final`. If you aren't sure whether a local variable is effectively final, just add the `final` keyword. If the code still compiles, the variable is effectively final.

Given this definition, which of the following variables are effectively final?

```
11: public String zooFriends() {
12:    String name = "Harry the Hippo";
13:    var size = 10;
14:    boolean wet;
15:    if(size> 100) size++;
16:    name.substring(0);
17:    wet = true;
18:    return name;
19: }
```

| Modifier | Description | Chapter Covered |
|---|---|---|
| `final` | Specifies that the instance variable must be initialized with each instance of the class exactly once | Chapter 5 |
| `volatile` | Instructs the JVM that the value in this variable may be modified by other threads | Chapter 13 |
| `transient` | Used to indicate that an instance variable should not be serialized with the class | Chapter 14 |

In Chapter 1, we show that instance variables receive default values based on their type when not set. For example, `int` receives a default value of `0`, while an object reference receives a default value of `null`. The compiler does not apply a default value to `final` variables, though. A `final` instance or `final static` variable must receive a value when it is declared or as part of initialization.

## Varargs

#### Rules for Creating a Method with a Varargs Parameter

1. A method can have at most one varargs parameter.
2. If a method contains a varargs parameter, it must be the last parameter in the list.

```
public class VisitAttractions {
   public void walk1(int… steps) {}
   public void walk2(int start, int… steps) {}
   public void walk3(int… steps, int start) {}     // DOES NOT COMPIL
   public void walk4(int… start, int… steps) {}  // DOES NOT COMPILE
}
```

```
import java.util.List;
import static java.util.Arrays.asList;          // static import
public class ZooParking {
   public static void main(String[] args) {
      List<String> list = asList("one", "two"); // No Arrays. prefix
   }
}
```

## Calling varargs methods with varargs.

```
// Pass an array
int[] data = new int[] {1, 2, 3};
walk1(data);

// Pass a list of values
walk1(1,2,3);
```

| | private | package | protected | public |
|---|---|---|---|---|
| the same class | Yes | Yes | Yes | Yes |
| another class in the same package | No | Yes | Yes | Yes |
| a subclass in a different package | No | No | Yes | Yes |
| an unrelated class in a different package | No | No | No | Yes |

```
Long badGorilla = 8;  // DOES NOT COMPILE
```

The compiler will automatically cast or autobox the `int` value to `long` or `Integer`, respectively. Neither of these types can be assigned to a `Long` reference variable, though, so the code does not compile. Compare this behavior to the previous example with `ears`, where the unboxed primitive value could be implicitly cast to a larger primitive type.

The types have to be compatible, though, as shown in the following examples.

```
Integer[] winterHours = { 10.5, 17.0 };  // DOES NOT COMPILE
Double[] summerHours  = { 9, 21 };       // DOES NOT COMPILE
```

```
public class Chimpanzee {
   public void climb(long t) {}
   public void swing(Integer u) {}
   public void jump(int v) {}
   public static void main(String[] args) {
      var c = new Chimpanzee();
      c.climb(123);
      c.swing(123);
      c.jump(123L);  // DOES NOT COMPILE
   }
}
```

## Overloading

```
public class Eagle {
   public void fly(int numMiles) {}
   public int fly(int numMiles) { return 1; }     // DOES NOT COMPILE
}
```

```
public class Hawk {
   public void fly(int numMiles) {}
   public static void fly(int numMiles) {}     // DOES NOT COMPILE
   public void fly(int numKilometers) {}       // DOES NOT COMPILE
}
```

This method doesn't compile because it differs from the original only by return type. The method signatures are the same, so they are duplicate methods as far as Java is concerned.

```
public class Glider {
   public static String glide(String s) {
      return "1";
   }
   public static String glide(String... s) {
      return "2";
   }
   public static String glide(Object o) {
      return "3";
   }
   public static String glide(String s, String t) {
      return "4";
   }
   public static void main(String[] args) {
      System.out.print(glide("a"));
      System.out.print(glide("a", "b"));
      System.out.print(glide("a", "b", "c"));
   }
}
```

It prints out `142`. The first call matches the signature taking a single `String` because that is the most specific match. The second call matches the signature taking two `String` parameters since that is an exact match. It isn't until the third call that the varargs version is used since there are no better matches.

## EXAM

1. A, E
2. b,c,
3. a, d,
4. a,b,c,e
5. a,c,d
6. a,b,d,f

6. Which of the following methods compile? (Choose all that apply.)
   A. `public void violin(int... nums) {}`
   B. `public void viola(String values, int... nums) {}`
   C. `public void cello(int... nums, String values) {}`
   D. `public void bass(String... values, int... nums) {}`
   E. `public void flute(String[] values, ...int nums) {}`
   F. `public void oboe(String[] values, int[] nums) {}`

Option D is incorrect because two varargs parameters are not allowed in the same method. Option E is incorrect because the ... for a varargs must be after

6. A, B, F. Options A and B are correct because the single varargs parameter is the last parameter declared. Option F is correct because it doesn't use any varargs parameters. Option C is incorrect because the varargs parameter is not last. Option D is incorrect because two varargs parameters are not allowed in the same method. Option E is incorrect because the ... for a varargs must be after the type, not before it.

7. b,c,d,e,f

7. Given the following method, which of the method calls return `2`? (Choose all that apply.)

```
public int juggle(boolean b, boolean... b2) {
   return b2.length;
}
```

A. `juggle();`
B. `juggle(true);`

```
Java
class Dog {
   String name;
```

```
public int juggle(boolean b, boolean… b2) {
   return b2.length;
}
```

A. `juggle();`
B. `juggle(true);`
C. `juggle(true, true);`
D. `juggle(true, true, true);`
E. `juggle(true, {true, true});`
F. `juggle(true, new boolean[2]);`

7. D, F. Options D and F are correct. Option D passes the initial parameter plus two more to turn into a varargs array of size 2. Option F passes the initial parameter plus an array of size 2. Option A does not compile because it does not pass the initial parameter. Option E does not compile because it does not declare an array properly. It should be `new boolean[] {true, true}`. Option B creates a varargs array of size 0, and option C creates a varargs array of size 1.

8. D.
9. c,b,d,f
10. B
11. b,e
12. B
13. B D

13. What is the output of the following code?

```
// RopeSwing.java
import rope.*;
import static rope.Rope.*;
public class RopeSwing {
   private static Rope rope1 = new Rope();
   private static Rope rope2 = new Rope();
   {
      System.out.println(ropes.length);
   }
   public static void main(String[] args) {
      rope1.length = 2;
      rope2.length = 8;
      System.out.println(ropes.length);
   }
}

// Rope.java
package rope;
public class Rope {
   public static int length = 0;
}
```

A. `0`
B. `00`
C. `2`
D. `8`
E. The code does not compile.
F. An exception is thrown.

13. D. There are two details to notice in this code. First, note that `RopeSwing` has an instance initializer and not a `static` initializer. Since `RopeSwing` is never constructed, the instance initializer does not run. The other detail is that `length` is `static`. Changes from any object update this common `static` variable. The code prints `0`, making option D correct.

14. E
15. a,b

15. Which of the following can replace line 2 to make this code compile?

```
1: import java.util.*;
2: // INSERT CODE HERE
3: public class Imports {
4:    public void method(ArrayList<String> list) {
5:       sort(list);
6:    }
7: }
```

A. `import static java.util.Collections;`
B. `import static java.util.Collections.*;`
C. `import static java.util.Collections.sort(ArrayList<String>);`
D. `static import java.util.Collections;`
E. `static import java.util.Collections.*;`
F. `static import java.util.Collections.sort(ArrayList<String>);`

15. B. The two valid ways to do this are `import static java.util.Collections.*;` and `import static java.util.Collections.sort;`, making option B correct. Option A is incorrect because you can do a static import only on `static` members. Classes such as `Collections` require a regular `import`. Option C is nonsense as method parameters have no business in an import. Options D, E, and F try to trick you into reversing the syntax of `import static`.

16. e
17. b
18. a,d B E

18. Which of the following are output by the following code? (Choose all that apply.)

```
public class StringBuilders {
   public static StringBuilder work(StringBuilder a,
      StringBuilder b) {
      a = new StringBuilder("a");
      b.append("b");
      return a;
   }
   public static void main(String[] args) {
      var s1 = new StringBuilder("s1");
      var s2 = new StringBuilder("s2");
      var s3 = work(s1, s2);
      System.out.println("s1 = " + s1);
      System.out.println("s2 = " + s2);
      System.out.println("s3 = " + s3);
   }
}
```

A. `s1 = a`
B. `s1 = s1`
C. `s2 = s2`
D. `s2 = s2b`
E. `s3 = a`
F. The code does not compile.

19. b,c,e,f

19. Which of the following will compile when independently inserted in the following code? (Choose all that apply.)

```
1:  public class Order3 {
2:     final String value1 = "red";
3:     static String value2 = "blue";
4:     String value3 = "yellow";
5:     {
6:        // CODE SNIPPET 1
7:     }
8:     static {
9:        // CODE SNIPPET 2
10:    } }
```

A. Insert at line 6: `value1 = "green";`
B. Insert at line 6: `value2 = "purple";`
C. Insert at line 6: `value3 = "orange";`
D. Insert at line 9: `value1 = "magenta";`
E. Insert at line 9: `value2 = "cyan";`
F. Insert at line 9: `value3 = "turquoise";`

20. a,e,
21. b,b

21. Which method signatures are valid overloads of the following method signature? (Choose all that apply.)

```
public void moo(int m, int… n)
```

Java

```
class Dog {
    String name;
}

public void rename(Dog d) {
    d.name = "Rex";          // modifies the original object
    d = new Dog();           // creates a new object
    d.name = "Max";          // modifies the new object
}
```

And you call:

Java

```
Dog myDog = new Dog();
myDog.name = "Buddy";
rename(myDog);
System.out.println(myDog.name); // Output: Rex
```

**But what happens at runti**

Let's say you call:

Java

```
moo(1);          // One int
moo(1, 2);       // Two ints
moo(1, 2, 3);    // Three ints
```

Then:

- `moo()` matches **both**:
  - `moo(int m, int… n)` → `m = 1`, `n = {}` (empty array)
  - `moo(int… x)` → `x = {1}`
- `moo(1, 2)` matches both:
  - `moo(int m, int… n)` → `m = 1`, `n = {2}`
  - `moo(int… x)` → `x = {1, 2}`
- `moo(1, 2, 3)` matches both:
  - `moo(int m, int… n)` → `m = 1`, `n = {2, 3}`
  - `moo(int… x)` → `x = {1, 2, 3}`

**So how does the compiler choose?**

Java uses **most specific match**:

- `moo(int m, int… n)` is **more specific** than `moo(int… x)` because it requires **at least one fixed argument**.
- So in all cases, the compiler will prefer `moo(int m, int… n)`.

21. Which method signatures are valid overloads of the following method signature? (Choose all that apply.)

```
public void moo(int m, int… n)
```

A. `public void moo(int a, int… b)`
B. `public int moo(char ch)`
C. `public void moooo(int… z)`
D. `private void moo(int… x)`
E. `public void moooo(int y)`
F. `public void moo(int… c, int d)`
G. `public void moo(int… i, int j…)`

21. B, D. Option A is incorrect because it has the same parameter list of types and therefore the same signature as the original method. Options B and D are the correct answers, as they are valid method overloads in which the types of parameters change. When overloading methods, the return type and access modifiers do not need to be the same. Options C and E are incorrect because the method name is different. Options F and G do not compile. There can be at most one varargs parameter, and it must be the last element in the parameter list.