

3.9 this Keyword

Self-Study Guide · About 50 minutes · With answer key

By the end you will be able to

- Explain what the keyword `this` holds inside an instance method or a constructor, and work out which object it refers to for a given call.
- Use `this` to reach an instance variable whose name is hidden by a parameter or local variable with the same name.
- Trace code that passes `this` as an argument so another method can work with the current object.
- Explain why a class method has no `this`, and fix code that uses `this`, or an instance member, where there is no current object.

What to remember from earlier

- You call an instance method on an object with a dot, and it works with that object's instance variables. ([Topic 3.5: Methods: How to Write Them, 3.5.1 Writing a Void Method in a Class](#))
- An object reference parameter is a copy of the reference, so a method can change the original object. ([Topic 3.6: Methods: Passing and Returning References of an Object, 3.6.1 Passing an Object Reference](#))
- A method may use the private data of a parameter of its own class. ([Topic 3.6: Methods: Passing and Returning References of an Object, 3.6.5 Private Data of a Parameter](#))
- A static method belongs to the class and is called with the class name. ([Topic 3.7: Class Variables and Methods, 3.7.4 Writing Class Methods](#))
- A parameter or local variable named like an instance variable hides it inside the method. ([Topic 3.8: Scope and Access, 3.8.6 When a Local Name Matches an Instance Variable](#))
- `==` on two references is true only when they refer to the very same object. ([Topic 2.6: Comparing Boolean Expressions, 2.6.4 Aliases and Reference Comparison](#))

3.9.1 this Refers to the Current Object

You have written one class and then created several objects from it. Each object has its own copy of every instance variable (Topic 3.3). But the code for a method is written only once, inside the class. When that one piece of code says `seat`, how does Java know **whose** `seat` it means? And what if a method needs to talk about "the object I belong to" as a whole, not just one of its variables?

The answer starts with how you call an instance method. In Topic 3.5 you wrote calls such as `hit.report()`. The variable before the dot holds a reference to one particular object, and the method runs **for that object**. While it runs, that object is the one in charge, like the person holding the microphone at a meeting. Anyone at the meeting can say "I", and "I" always means whoever is holding the microphone right now.

Java gives every instance method and every constructor a built-in way to say "I". That way is the keyword **this**. Think of it as a variable you never declare, which stores a reference to the **current object**: whichever object the running method was called on, or the new object a running constructor is setting up. You also never assign to it. Java fills it in automatically at the start of each call with the reference that was used to make the call. (Here the microphone analogy stops working: at a meeting people grab the microphone whenever they like, but in Java the call itself decides who holds it, every single time.)

KEY IDEA

Inside an instance method or a constructor, `this` holds a reference to the current object, the object the method was called on or the object the constructor is building. `this.seat` means "the `seat` instance variable of the current object."

Here is the syntax taken apart:

Piece	What it is
<code>this</code>	A keyword (always lowercase) that holds a reference to the current object
<code>.</code>	The dot operator, the same one you use in <code>hit.report()</code>
<code>seat</code>	An instance variable or method of the current object
<code>this.seat</code>	The current object's <code>seat</code> , exactly like <code>front.seat</code> would be for <code>front</code>

WORKED EXAMPLE Which seat does `this.seat` mean?

The `Ticket` class below stores one seat code. Its `describe` method prints `this.seat`. Watch which object `this` refers to during each call. The helper class `Ticket` comes first in the file, and the class with `main` comes after it.

SeatTicket.java

```

1  class Ticket
2  {
3      private String seat;
4
5      public Ticket(String seatCode)
6      {
7          seat = seatCode;
8      }
9
10     public void describe()
11     {
12         System.out.println("Seat " + this.seat);
13     }
14 }
15
16 public class SeatTicket
17 {
18     public static void main(String[] args)
19     {
20         Ticket front = new Ticket("A1");
21         Ticket back = new Ticket("K9");
22         back.describe();
23         front.describe();
24     }
25 }

```

Lines 20 and 21 build two objects. Picture the reference in each variable as an address:

Variable	Holds	Object at that address
front	address A	Ticket with seat A1
back	address B	Ticket with seat K9

Now follow the two calls:

Line	Call	this holds	this.seat	Printed
22	back.describe()	address B	"K9"	Seat K9
23	front.describe()	address A	"A1"	Seat A1

```

Seat K9
Seat A1

```

On line 22 Java copies the reference in back (address B) into `this`, then runs line 12. `this.seat` follows address B and finds "K9". On line 23 the very same line 12 runs again, but now `this` holds address A, so it finds "A1". One method, two calls, two different current objects.

The table above claims that `this` holds the same reference as the variable in front of the dot. You can test that claim directly. In Topic 2.6 you learned that `==` on two references is `true` only when they refer to the very same object. The next class compares `this` with a parameter.

WORKED EXAMPLE Testing what this holds

SameTicket.java

```

1  class Ticket
2  {
3      private String seat;
4
5      public Ticket(String seatCode)
6      {
7          seat = seatCode;
8      }
9
10     public boolean isSameAs(Ticket other)
11     {
12         return this == other;
13     }
14 }
15
16 public class SameTicket
17 {
18     public static void main(String[] args)
19     {
20         Ticket front = new Ticket("A1");
21         Ticket back = new Ticket("K9");
22         Ticket myTicket = back;
23         System.out.println(front.isSameAs(front));
24         System.out.println(front.isSameAs(back));
25         System.out.println(back.isSameAs(myTicket));
26     }
27 }

```

Line 22 makes `myTicket` an alias of `back`: both hold address B. Now trace each call to `isSameAs`, remembering that `this` comes from the variable before the dot and `other` comes from the argument in the parentheses.

Line	Call	this	other	this == other
23	<code>front.isSameAs(front)</code>	address A	address A	true
24	<code>front.isSameAs(back)</code>	address A	address B	false
25	<code>back.isSameAs(myTicket)</code>	address B	address B	true

```

true
false
true

```

Line 25 is the interesting one. The argument is `myTicket`, a different variable from `back`, but it holds the same address, so inside the method `this` and `other` refer to one object and `==` is true. The expression `this == other` is **self-referencing**: it asks the current object to compare itself with something else.

You may have noticed that the `Ticket` constructor on line 7 writes plain `seat`, not `this.seat`, and it works. That is because Java quietly treats a plain instance variable name inside an instance

method as `this.seat`. Writing `this.out` is optional when nothing else has the same name. You will see in 3.9.2 when it stops being optional.

WATCH OUT

Thinking this means the newest object, or the class. Some students picture `this` as "the object that was created most recently". In `SeatTicket.java`, `back` was created last (line 21), yet line 23 printed `Seat A1`, because the call was `front.describe()`. Other students picture `this` as the `Ticket` class itself, but a class has no single `seat` value to print; each object has its own.

`this` is decided fresh at every call, by the reference before the dot. To find what `this` refers to, look at the call that started the method, not at the order in which objects were created.

PAUSE AND TRY

Question 1 Tracing

Trace this program. Write the exact output and the value of `this.steps` at the end of each call to `walk`.

`StepCounter.java`

```
1 class Pedometer
2 {
3     private int steps;
4
5     public Pedometer(int start)
6     {
7         steps = start;
8     }
9
10    public void walk(int amount)
11    {
12        this.steps = this.steps + amount;
13    }
14
15    public int getSteps()
16    {
17        return this.steps;
18    }
19 }
20
21 public class StepCounter
22 {
23     public static void main(String[] args)
24     {
25         Pedometer mia = new Pedometer(100);
26         Pedometer leo = new Pedometer(50);
27         leo.walk(30);
28         mia.walk(5);
29         leo.walk(10);
30         System.out.println(mia.getSteps());
31         System.out.println(leo.getSteps());
```

```
32     }  
33 }
```

Question 2 Multiple choice

Using the `Ticket` class from `SameTicket.java`, a program runs `back.isSameAs(front)`. While `isSameAs` runs, what does `this` hold?

- A. A reference to the object `front` refers to
- B. A reference to the object `back` refers to
- C. A reference to whichever `Ticket` was created most recently
- D. Nothing; this is only filled in inside constructors

3.9.2 Reaching a Hidden Instance Variable

In Topic 3.8 you learned a rule about names: when a parameter or local variable has the same name as an instance variable, that name means the parameter or local variable inside the method. Programmers like to give constructor parameters the same names as the instance variables they fill, because `title` is the clearest name for both. But the 3.8 rule then gets in the way. Here is what happens:

`ShadowBug.java`

```
1  class Song  
2  {  
3      private String title;  
4      private int plays;  
5  
6      public Song(String title, int plays)  
7      {  
8          title = title;  
9          plays = plays;  
10     }  
11  
12     public String getTitle()  
13     {  
14         return title;  
15     }  
16  
17     public int getPlays()  
18     {  
19         return plays;  
20     }  
21 }  
22  
23 public class ShadowBug  
24 {  
25     public static void main(String[] args)  
26     {  
27         Song hit = new Song("Sunrise", 12);  
28         System.out.println(hit.getTitle());  
29     }  
30 }
```

```
29         System.out.println(hit.getPlays());
30     }
31 }
```

```
null
0
```

The program compiles and runs, but the song has no title and no plays. On line 8, both names `title` mean the **parameter** `title`, so the line copies the parameter into itself. The same thing happens to `plays` on line 9. The instance variables are never touched, so they keep their default values from Topic 3.4: `null` for a `String` and `0` for an `int`.

When a local name hides an instance variable like this, we say the instance variable is **shadowed**. A plain name can no longer reach it. But `this` can: `this.title` always means "the current object's `title`", because the dot tells Java to look inside an object, and parameters do not live inside objects.

KEY IDEA

When a parameter or local variable has the same name as an instance variable, use `this.name` for the instance variable and the plain `name` for the parameter. `this.title = title;` copies the parameter into the current object's instance variable.

WORKED EXAMPLE Fixing the constructor with this

The only change from `ShadowBug.java` is lines 8 and 9:

`ShadowFixed.java`

```
1  class Song
2  {
3      private String title;
4      private int plays;
5
6      public Song(String title, int plays)
7      {
8          this.title = title;
9          this.plays = plays;
10     }
11
12     public String getTitle()
13     {
14         return title;
15     }
16
17     public int getPlays()
18     {
19         return plays;
20     }
21 }
22
23 public class ShadowFixed
24 {
```

```

25     public static void main(String[] args)
26     {
27         Song hit = new Song("Sunrise", 12);
28         System.out.println(hit.getTitle());
29         System.out.println(hit.getPlays());
30     }
31 }

```

Inside a constructor, the current object is the one being built. Follow line 27, `new Song("Sunrise", 12)`:

Step	What Java does	this.title	this.plays
1	<code>new</code> makes a new object with default values; <code>this</code> refers to it	null	0
2	Line 8: right side <code>title</code> is the parameter <code>"Sunrise"</code> ; store it in the object	"Sunrise"	0
3	Line 9: right side <code>plays</code> is the parameter <code>12</code> ; store it in the object	"Sunrise"	12
4	The constructor ends; <code>new</code> hands the object's reference to <code>hit</code>	"Sunrise"	12

```

Sunrise
12

```

The getters on lines 14 and 19 write plain `title` and `plays`. Nothing shadows those names inside the getters, so they mean the instance variables, just as if they said `this.title` and `this.plays`.

Shadowing is not only a constructor problem. A method can have a parameter with an instance variable's name too, and then a single expression can use both. That is where `this` shows its value most clearly.

WORKED EXAMPLE Using both names in one expression

SongPlays.java

```

1  class Song
2  {
3      private String title;
4      private int plays;
5
6      public Song(String title, int plays)
7      {
8          this.title = title;
9          this.plays = plays;
10     }
11
12     public void addPlays(int plays)
13     {
14         this.plays = this.plays + plays;
15     }
16
17     public void report()
18     {

```

```

19         System.out.println(title + ": " + plays);
20     }
21 }
22
23 public class SongPlays
24 {
25     public static void main(String[] args)
26     {
27         Song hit = new Song("Sunrise", 12);
28         hit.addPlays(5);
29         hit.report();
30         hit.addPlays(3);
31         hit.report();
32     }
33 }

```

Look at line 14 inside `addPlays`. The parameter is named `plays`, so the plain name `plays` is the amount to add, and `this.plays` is the song's running total:

Line	Call	this.plays before	parameter plays	this.plays after
28	<code>hit.addPlays(5)</code>	12	5	17
30	<code>hit.addPlays(3)</code>	17	3	20

```

Sunrise: 17
Sunrise: 20

```

In `report` (line 19) there is no parameter, so plain `plays` means the instance variable. The same name means different things in different methods, and `this.` removes any doubt.

WATCH OUT

Putting `this` on the wrong side. A student remembers that the constructor needs `this` but writes it on the right:

ReversedThis.java

```

1 class Song
2 {
3     private String title;
4     private int plays;
5
6     public Song(String title, int plays)
7     {
8         title = this.title;
9         this.plays = plays;
10    }
11
12    public void addPlays(int plays)
13    {
14        this.plays = this.plays + plays;
15    }
16

```

```
17     public void report()
18     {
19         System.out.println(title + ": " + plays);
20     }
21 }
22
23 public class ReversedThis
24 {
25     public static void main(String[] args)
26     {
27         Song hit = new Song("Sunrise", 12);
28         hit.addPlays(5);
29         hit.report();
30         hit.addPlays(3);
31         hit.report();
32     }
33 }
```

```
null: 17
null: 20
```

Line 8 reads the instance variable (still `null`) and stores it in the parameter, which is thrown away when the constructor ends. Assignment copies from right to left (Topic 1.4), so the side that **receives** the value is the left one. The instance variable must be on the left:

```
this.title = title;.
```

PAUSE AND TRY

Question 3 Debugging

This program prints `null is at level 3` instead of `Zara is at level 3`. Find the line that causes the problem, explain why, and write the corrected line.

PlayerBug.java

```
1  class Player
2  {
3      private String name;
4      private int level;
5
6      public Player(String name, int level)
7      {
8          name = name;
9          this.level = level;
10     }
11
12     public void show()
13     {
14         System.out.println(name + " is at level " + level);
15     }
16 }
17
18 public class PlayerBug
19 {
20     public static void main(String[] args)
21     {
22         Player hero = new Player("Zara", 3);
23         hero.show();
24     }
25 }
```

Question 4 Multiple choice

What does this program print?

ThermostatRaise.java

```
1 class Thermostat
2 {
3     private int target;
4
5     public Thermostat(int target)
6     {
7         this.target = target;
8     }
9
10    public void raise(int amount)
11    {
12        int target = this.target + amount;
13        System.out.println("New target: " + target);
14    }
15
16    public int getTarget()
17    {
18        return target;
19    }
20 }
21
22 public class ThermostatRaise
23 {
24     public static void main(String[] args)
25     {
26         Thermostat hall = new Thermostat(68);
27         hall.raise(4);
28         System.out.println(hall.getTarget());
29     }
30 }
```

- A. New target: 72 then 72
- B. New target: 72 then 68
- C. New target: 68 then 68
- D. Nothing; a local variable cannot have the same name as an instance variable

3.9.3 Passing this as an Argument

So far `this` has been a way to reach the current object's own variables. Sometimes a method needs to hand **the whole current object** to someone else. Think of a pet going to the vet. The pet's `visit` method needs to call the vet's `examine` method, and `examine` needs to know which pet it is examining. Inside `visit`, there is no variable named `rex` or `luna`; the method does not know what the caller named the object. What it does have is `this`.

Recall from Topic 3.6 that when an argument is a reference, the parameter receives a copy of that reference, not a copy of the object. So passing `this` gives the other method a reference to the current object itself. Whatever that method does through its parameter happens to the current object.

KEY IDEA

Writing `this` as an argument, as in `doctor.examine(this)`, passes a reference to the current object, so the called method can read or change that same object.

WORKED EXAMPLE A pet visits the vet

VetVisit.java

```
1  class Vet
2  {
3      private String vetName;
4
5      public Vet(String vetName)
6      {
7          this.vetName = vetName;
8      }
9
10     public void examine(Pet patient)
11     {
12         System.out.println(vetName + " examines " + patient.getName());
13         patient.markHealthy();
14     }
15 }
16
17 class Pet
18 {
19     private String name;
20     private boolean healthy;
21
22     public Pet(String name)
23     {
24         this.name = name;
25         healthy = false;
26     }
27
28     public String getName()
29     {
30         return name;
31     }
32
33     public boolean isHealthy()
34     {
35         return healthy;
36     }
37
38     public void markHealthy()
39     {
40         healthy = true;
41     }
42
43     public void visit(Vet doctor)
44     {
45         doctor.examine(this);
```

```

46     }
47 }
48
49 public class VetVisit
50 {
51     public static void main(String[] args)
52     {
53         Pet rex = new Pet("Rex");
54         Pet luna = new Pet("Luna");
55         Vet ortiz = new Vet("Dr. Ortiz");
56         rex.visit(ortiz);
57         System.out.println(rex.isHealthy());
58         System.out.println(luna.isHealthy());
59     }
60 }

```

Two classes work together. `Vet.examine` takes a `Pet` parameter named `patient`, prints its name, and marks it healthy. `Pet.visit` takes a `Vet` and calls `examine`, passing this. Suppose `rex` holds address R, `luna` holds address L, and `ortiz` holds address V.

Step	Line	What happens	rex healthy?
1	56	<code>rex.visit(ortiz)</code> : <code>this = R</code> , <code>doctor = V</code>	false
2	45	<code>doctor.examine(this)</code> : calls <code>examine</code> on V and passes R	false
3	12	Inside <code>examine</code> , <code>this = V</code> and <code>patient = R</code> ; prints the line	false
4	13	<code>patient.markHealthy()</code> : calls <code>markHealthy</code> on R	false
5	40	Inside <code>markHealthy</code> , <code>this = R</code> ; sets R's healthy to true	true
6	57	Back in <code>main</code> , prints <code>rex.isHealthy()</code>	true
7	58	Prints <code>luna.isHealthy()</code> ; L was never touched	true

```

Dr. Ortiz examines Rex
true
false

```

Notice how `this` changes as the calls nest. In `visit` it is the pet R. In `examine` it is the vet V, because `examine` was called on `doctor`. In `markHealthy` it is R again, because that call was made on `patient`. Each method has its own `this`, set by its own call. The reference R traveled from `this` in `visit` into the parameter `patient`, which is how the vet changed Rex and only Rex.

The other method does not have to be in a different class. Here a `Rectangle` passes itself to another `Rectangle`, so one comparison method can be reused to build a second one.

WORKED EXAMPLE Asking the other object to compare with me

RectangleCompare.java

```

1 class Rectangle
2 {
3     private int width;
4     private int height;
5

```

```

6     public Rectangle(int width, int height)
7     {
8         this.width = width;
9         this.height = height;
10    }
11
12    public int area()
13    {
14        return width * height;
15    }
16
17    public boolean isLargerThan(Rectangle other)
18    {
19        return this.area() > other.area();
20    }
21
22    public boolean isSmallerThan(Rectangle other)
23    {
24        return other.isLargerThan(this);
25    }
26 }
27
28 public class RectangleCompare
29 {
30     public static void main(String[] args)
31     {
32         Rectangle poster = new Rectangle(4, 6);
33         Rectangle card = new Rectangle(3, 5);
34         System.out.println(card.isSmallerThan(poster));
35         System.out.println(poster.isSmallerThan(card));
36     }
37 }

```

isLargerThan (line 19) compares the current rectangle's area with another's. isSmallerThan (line 24) does not redo that work. "I am smaller than you" means the same as "you are larger than me", so it calls isLargerThan **on the other rectangle** and passes this as the argument. Trace line 34, where poster has area $4 \times 6 = 24$ and card has area $3 \times 5 = 15$:

Step	Line	Call	this	other	Result
1	34	card.isSmallerThan(poster)	card	poster	
2	24	other.isLargerThan(this)	poster	card	
3	19	this.area() > other.area()	poster	card	24 > 15: true

Line 35 is the reverse: poster.isSmallerThan(card) becomes card.isLargerThan(poster), and $15 > 24$ is false.

```

true
false

```

In step 2 the roles swap: the object that was `this` in `isSmallerThan` becomes `other` in `isLargerThan`, and the old `other` becomes the new `this`. Line 19 also shows that `this.area()` calls a method on the current object; plain `area()` would mean the same thing.

WATCH OUT

Passing a new object instead of `this`. A student does not trust `this` and builds a pet to send instead:

VisitCopyBug.java

```
1  class Vet
2  {
3      private String vetName;
4
5      public Vet(String vetName)
6      {
7          this.vetName = vetName;
8      }
9
10     public void examine(Pet patient)
11     {
12         System.out.println(vetName + " examines " + patient.getName());
13         patient.markHealthy();
14     }
15 }
16
17 class Pet
18 {
19     private String name;
20     private boolean healthy;
21
22     public Pet(String name)
23     {
24         this.name = name;
25         healthy = false;
26     }
27
28     public String getName()
29     {
30         return name;
31     }
32
33     public boolean isHealthy()
34     {
35         return healthy;
36     }
37
38     public void markHealthy()
39     {
40         healthy = true;
41     }
42
43     public void visit(Vet doctor)
```

```
44     {
45         doctor.examine(new Pet(name));
46     }
47 }
48
49 public class VisitCopyBug
50 {
51     public static void main(String[] args)
52     {
53         Pet rex = new Pet("Rex");
54         Pet luna = new Pet("Luna");
55         Vet ortiz = new Vet("Dr. Ortiz");
56         rex.visit(ortiz);
57         System.out.println(rex.isHealthy());
58         System.out.println(luna.isHealthy());
59     }
60 }
```

```
Dr. Ortiz examines Rex
false
false
```

Line 45 now passes `new Pet(name)`, a brand-new object that happens to have the same name. The vet prints the right name, so the bug is easy to miss, but `markHealthy` changes the new object, which nobody keeps a reference to. `rex`'s object is still unhealthy. To let another method act on the current object, pass `this`, which is a reference to that object and not a copy of it.

PAUSE AND TRY

Question 5 Tracing

Write the exact output of this program.

FriendWave.java

```
1 class Friend
2 {
3     private String name;
4
5     public Friend(String name)
6     {
7         this.name = name;
8     }
9
10    public String getName()
11    {
12        return name;
13    }
14
15    public void greet(Friend other)
16    {
```

```

17         System.out.println(name + " waves at " + other.getName());
18         other.waveBack(this);
19     }
20
21     public void waveBack(Friend greeter)
22     {
23         System.out.println(name + " waves back at " + greeter.getName());
24     }
25 }
26
27 public class FriendWave
28 {
29     public static void main(String[] args)
30     {
31         Friend kai = new Friend("Kai");
32         Friend noor = new Friend("Noor");
33         noor.greet(kai);
34         kai.greet(noor);
35     }
36 }

```

Question 6 Short answer

A student says, "In VetVisit.java, doctor.examine(this) sends the vet a copy of the pet, so the vet's changes should not affect rex." Explain why the student is wrong.

3.9.4 Class Methods Have No this

In Topic 3.7 you wrote **class methods**, the methods marked `static`. They belong to the class as a whole and are usually called with the class name, as in `Math.sqrt(16.0)` or `Ticket.getTicketsSold()`. Ask the 3.9.1 question about such a call: which object is in charge while it runs? Look at the call. There is no object before the dot, only a class name. No object, no current object, and so nothing for `this` to hold.

That is why Java gives a class method **no this at all**. It also explains a rule you met in Topic 3.7: a class method cannot use instance variables or call instance methods directly. A plain instance variable name means `this.name`, and an instance method call with no dot means `this.method()`. Without a `this`, neither has an object to work on.

KEY IDEA

A class (static) method runs without any current object, so it has no `this`. Using `this`, or a plain instance variable or instance method name, inside a class method is a compile error.

WORKED EXAMPLE Asking a class method for this.seat

This version of `Ticket` makes `getSeat` static by mistake:

StaticThisError.java

```
1 class Ticket
2 {
3     private String seat;
4
5     public Ticket(String seat)
6     {
7         this.seat = seat;
8     }
9
10    public static String getSeat()
11    {
12        return this.seat;
13    }
14 }
15
16 public class StaticThisError
17 {
18     public static void main(String[] args)
19     {
20         Ticket front = new Ticket("A1");
21         System.out.println(Ticket.getSeat());
22     }
23 }
```

Java refuses to compile it:

```
StaticThisError.java:12: error: non-static variable this cannot be referenced from a
static context
    return this.seat;
           ^
1 error
```

The message calls `this` a "non-static variable", meaning it belongs to an object, and line 12 is in a "static context", a class method. Think about line 21: `Ticket.getSeat()` names no ticket. If there were three tickets, Java would have no way to choose whose seat to return. (Exact error wording can vary between Java versions; the key words are "static context".)

There are two ways to fix it, depending on what the method should do. If the method is about one ticket, remove `static` so it becomes an instance method, and call it on an object: `front.getSeat()`. If the method really belongs to the class, give it a parameter so the caller supplies the object, as the next example does.

The next example shows what a class method **can** do: use class variables, and work with objects that are passed to it as parameters.

WORKED EXAMPLE A class method that receives its objects

StaticWithParameter.java

```
1 class Ticket
2 {
3     private static int ticketsSold = 0;
```

```

4     private String seat;
5
6     public Ticket(String seat)
7     {
8         this.seat = seat;
9         ticketsSold++;
10    }
11
12    public String getSeat()
13    {
14        return this.seat;
15    }
16
17    public static int getTicketsSold()
18    {
19        return ticketsSold;
20    }
21
22    public static boolean sameRow(Ticket first, Ticket second)
23    {
24        return first.seat.substring(0, 1).equals(second.seat.substring(0, 1));
25    }
26 }
27
28 public class StaticWithParameter
29 {
30     public static void main(String[] args)
31     {
32         Ticket front = new Ticket("A1");
33         Ticket beside = new Ticket("A2");
34         Ticket back = new Ticket("K9");
35         System.out.println(Ticket.getTicketsSold());
36         System.out.println(Ticket.sameRow(front, beside));
37         System.out.println(Ticket.sameRow(front, back));
38         System.out.println(back.getSeat());
39     }
40 }

```

Line	What happens	Printed
32–34	Three constructors run. In each, <code>this</code> is the new ticket; line 9 adds 1 to the class variable	
35	<code>getTicketsSold</code> is a class method, but it uses only the class variable <code>ticketsSold</code> , which is 3	3
36	<code>sameRow(front, beside)</code> : <code>first.seat</code> is "A1", <code>second.seat</code> is "A2"; both start with "A"	true
37	<code>sameRow(front, back)</code> : "A" versus "K"	false
38	<code>getSeat</code> is an instance method called on <code>back</code> , so <code>this.seat</code> is "K9"	K9

```
3
true
false
K9
```

`sameRow` never mentions `this`. It reaches each ticket's seat through a parameter, `first.seat` and `second.seat`. (It may use the private `seat` directly because the parameters are `Tickets` and the method is inside the `Ticket` class, as in Topic 3.6.) The class method works with objects only when someone hands it references to them.

Here is the comparison side by side:

Question	Instance method	Class (static) method
How is it usually called?	On an object: <code>back.getSeat()</code>	On the class: <code>Ticket.getTicketsSold()</code>
Is there a current object?	Yes, the object before the dot	No
Can it use <code>this</code> ?	Yes	No; compile error
How does it reach an object's variables?	<code>this.seat</code> , or plain <code>seat</code>	Only through a parameter: <code>first.seat</code>

WATCH OUT

Using `this` in `main`. `main` is a class method too: its header says `public static void main`. So `this` does not exist there, even when `main` is written inside the class whose objects it creates:

MainThis.java

```
1 public class MainThis
2 {
3     private int score;
4
5     public MainThis(int score)
6     {
7         this.score = score;
8     }
9
10    public int getScore()
11    {
12        return this.score;
13    }
14
15    public static void main(String[] args)
16    {
17        MainThis game = new MainThis(40);
18        System.out.println(this.getScore());
19    }
20 }
```

```
MainThis.java:18: error: non-static variable this cannot be referenced from a static context
```

```
    System.out.println(this.getScore());
                        ^
```

```
1 error
```

Students write this because `main` sits inside the class, so it feels like `main` belongs to an object. It does not; it runs before any object exists. The fix is to call the method on the object that line 17 created: `System.out.println(game.getScore());`, which prints `40`.

PAUSE AND TRY

Question 7 Multiple choice

Each choice is a method that could be added to this class. Which one does **not** compile?

```
1 public class Lamp
2 {
3     private int brightness;
4
5     public Lamp(int brightness)
6     {
7         this.brightness = brightness;
8     }
9 }
```

- A. `public static void turnOff() { this.brightness = 0; }`
- B. `public static int maxBrightness() { return 100; }`
- C. `public void dim() { this.brightness = 0; }`
- D. `public static void dimLamp(Lamp lamp) { lamp.brightness = 0; }`

Question 8 Debugging

This program does not compile. Explain the error and write the corrected line.

MainCallBug.java

```
1 public class MainCallBug
2 {
3     private int score;
4
5     public MainCallBug(int score)
6     {
7         this.score = score;
8     }
9
10    public void printScore()
11    {
12        System.out.println("Score: " + score);
13    }
14
15    public static void main(String[] args)
16    {
17        MainCallBug game = new MainCallBug(40);
18        printScore();
19    }
20 }
```

```
MainCallBug.java:18: error: non-static method printScore() cannot be referenced from
a static context
    printScore();
    ^
```

Putting it together

This bank account class uses every idea from the chapter: `this.` to reach shadowed instance variables in the constructor, `this.balance` in self-referencing expressions, `this` passed as an argument, and a class method that has no `this`. Before reading the trace, predict all five lines of output.

WORKED EXAMPLE Transfers between two accounts

BankTransfer.java

```
1 class BankAccount
2 {
3     private static int accountsOpened = 0;
4     private String owner;
5     private int balance;
6
7     public BankAccount(String owner, int balance)
8     {
9         this.owner = owner;
10        this.balance = balance;
11        accountsOpened++;
12    }
```

```

13
14     public String getOwner()
15     {
16         return owner;
17     }
18
19     public int getBalance()
20     {
21         return balance;
22     }
23
24     public void transferTo(BankAccount other, int amount)
25     {
26         this.balance = this.balance - amount;
27         other.receive(this, amount);
28     }
29
30     public void receive(BankAccount sender, int amount)
31     {
32         this.balance = this.balance + amount;
33         System.out.println(owner + " got " + amount + " from " +
sender.getOwner());
34     }
35
36     public static int getAccountsOpened()
37     {
38         return accountsOpened;
39     }
40 }
41
42 public class BankTransfer
43 {
44     public static void main(String[] args)
45     {
46         BankAccount ana = new BankAccount("Ana", 100);
47         BankAccount ben = new BankAccount("Ben", 20);
48         ana.transferTo(ben, 30);
49         ben.transferTo(ana, 5);
50         System.out.println(ana.getBalance());
51         System.out.println(ben.getBalance());
52         System.out.println(BankAccount.getAccountsOpened());
53     }
54 }

```

Lines 46 and 47 run the constructor twice. In each run, `this` is the account being built, so lines 9 and 10 fill that account's `owner` and `balance` from the same-named parameters, and line 11 adds 1 to the shared class variable. Now the transfers:

Step	Line	Running	this	Other names	Ana's balance	Ben's balance
0	47	(after both constructors)			100	20
1	48	<code>ana.transferTo(ben, 30)</code>	Ana	<code>other = Ben</code>	100	20
2	26	<code>this.balance - amount</code>	Ana		70	20

3	27	<code>other.receive(this, 30)</code>	Ana	passes Ana	70	20
4	32	inside <code>receive</code>	Ben	<code>sender = Ana</code>	70	50
5	33	prints Ben got 30 from Ana	Ben	<code>sender = Ana</code>	70	50
6	49	<code>ben.transferTo(ana, 5)</code>	Ben	<code>other = Ana</code>	70	50
7	26	<code>this.balance - amount</code>	Ben		70	45
8	32	inside <code>receive</code>	Ana	<code>sender = Ben</code>	75	45
9	33	prints Ana got 5 from Ben	Ana	<code>sender = Ben</code>	75	45

Then line 50 prints 75, line 51 prints 45, and line 52 calls the class method `getAccountsOpened`, which returns the class variable, 2.

```
Ben got 30 from Ana
Ana got 5 from Ben
75
45
2
```

Three things to notice:

- In `receive`, the plain name `owner` on line 33 means `this.owner`, the account receiving money, while `sender.getOwner()` asks the account that was passed in. Passing `this` on line 27 is the only way `receive` learns who sent the money.
- `this` flips between the two accounts as the calls nest: in `transferTo` it is the sender, in `receive` it is the receiver.
- `getAccountsOpened` is called with the class name and has no `this`. It works because it uses only the class variable, never `owner` or `balance`.

Vocabulary

Term	Definition	Example
this	A keyword that, inside an instance method or constructor, holds a reference to the current object	<code>this.seat</code>
current object	The object whose instance method or constructor is running: the one the method was called on, or the one being built	<code>back</code> in <code>back.describe()</code>
shadowed	Describes an instance variable hidden by a parameter or local variable with the same name	<code>title</code> in <code>Song(String title, ...)</code>
self-referencing expression	An expression that uses <code>this</code> to refer to the current object	<code>this == other</code>
class method	A static method; it belongs to the class, is called with the class name, and has no <code>this</code>	<code>Ticket.getTicketsSold()</code>

Chapter summary

- Inside an instance method or a constructor, `this` holds a reference to the current object: the object before the dot in the call, or the object a constructor is building. It is set again at every call.
- A plain instance variable or instance method name inside an instance method means the same as writing `this.` in front of it.
- When a parameter or local variable shadows an instance variable, `this.name` reaches the instance variable; `this.title = title;` stores a parameter in the object. Forgetting `this.` compiles but leaves the instance variable unchanged.
- Passing `this` as an argument hands another method a reference to the current object, so that method can read or change the same object, not a copy.
- A class (static) method has no current object, so it has no `this`; using `this` or an instance member directly there, including in `main`, is a compile error. A class method can work with objects passed to it as parameters.

Review questions

Question 9 Multiple choice

Which statement about the keyword `this` is true?

- A. `this` refers to the object that was created most recently.
- B. Inside a constructor, `this` refers to the object that the constructor is building.
- C. `this` can be used inside any method, including `static` methods, as long as at least one object exists.
- D. In `this.score = score;`, the value of the instance variable is copied into the parameter.

Question 10 Tracing

The method `larger` returns a reference to whichever counter has the larger count. Trace the program and write the exact output.

LargerCounter.java

```
1 class Counter
2 {
3     private int count;
4
5     public Counter(int count)
6     {
7         this.count = count;
8     }
9
10    public Counter larger(Counter other)
11    {
12        if (this.count >= other.count)
13        {
14            return this;
15        }
16        return other;
17    }
```

```
18
19     public void increment()
20     {
21         count++;
22     }
23
24     public int getCount()
25     {
26         return count;
27     }
28 }
29
30 public class LargerCounter
31 {
32     public static void main(String[] args)
33     {
34         Counter red = new Counter(7);
35         Counter blue = new Counter(9);
36         Counter winner = red.larger(blue);
37         winner.increment();
38         Counter again = red.larger(red);
39         again.increment();
40         System.out.println(red.getCount());
41         System.out.println(blue.getCount());
42     }
43 }
```

Question 11 Debugging

A student changes line 45 of VetVisit.java and gets a compile error. Explain the error and write the corrected line.

ExamineWrongArg.java

```
1 class Vet
2 {
3     private String vetName;
4
5     public Vet(String vetName)
6     {
7         this.vetName = vetName;
8     }
9
10    public void examine(Pet patient)
11    {
12        System.out.println(vetName + " examines " + patient.getName());
13        patient.markHealthy();
14    }
15 }
16
17 class Pet
18 {
19     private String name;
20     private boolean healthy;
21 }
```

```
22     public Pet(String name)
23     {
24         this.name = name;
25         healthy = false;
26     }
27
28     public String getName()
29     {
30         return name;
31     }
32
33     public boolean isHealthy()
34     {
35         return healthy;
36     }
37
38     public void markHealthy()
39     {
40         healthy = true;
41     }
42
43     public void visit(Vet doctor)
44     {
45         doctor.examine(doctor);
46     }
47 }
48
49 public class ExamineWrongArg
50 {
51     public static void main(String[] args)
52     {
53         Pet rex = new Pet("Rex");
54         Pet luna = new Pet("Luna");
55         Vet ortiz = new Vet("Dr. Ortiz");
56         rex.visit(ortiz);
57         System.out.println(rex.isHealthy());
58         System.out.println(luna.isHealthy());
59     }
60 }
```

```
ExamineWrongArg.java:45: error: incompatible types: Vet cannot be converted to Pet
        doctor.examine(doctor);
                        ^
```

Question 12 Multiple choice

Using the `BankAccount` class from *Putting it together*, what does this code print?

```
1 BankAccount ben = new BankAccount("Ben", 20);
2 ben.transferTo(ben, 10);
3 System.out.println(ben.getBalance());
```

- A. Ben got 10 from Ben then 20
- B. Ben got 10 from Ben then 10

- C. Ben got 10 from Ben then 30
- D. Nothing; an object cannot be passed to its own method

Question 13 Tracing

Here a constructor passes `this` to another object. Write the exact output.

ClubJoin.java

```
1 class Club
2 {
3     private int members;
4     private String newest;
5
6     public Club()
7     {
8         members = 0;
9         newest = "nobody";
10    }
11
12    public void join(Student s)
13    {
14        members++;
15        newest = s.getName();
16    }
17
18    public void report()
19    {
20        System.out.println(members + " members, newest: " + newest);
21    }
22 }
23
24 class Student
25 {
26     private String name;
27
28     public Student(String name, Club club)
29     {
30         this.name = name;
31         club.join(this);
32     }
33
34     public String getName()
35     {
36         return name;
37     }
38 }
39
40 public class ClubJoin
41 {
42     public static void main(String[] args)
43     {
44         Club chess = new Club();
45         chess.report();
46         Student amir = new Student("Amir", chess);
47         Student bea = new Student("Bea", chess);
48         chess.report();
```

```
49     }  
50 }
```

Question 14 Short answer

A student adds this class method to `Ticket`:

```
1 public static boolean isFrontRow()  
2 {  
3     return this.seat.substring(0, 1).equals("A");  
4 }
```

Explain why it does not compile, and describe one way to change the method so that it does.

Reflection

- ☐ I can say which object `this` refers to for a given method call or constructor. Reread: 3.9.1 `this` Refers to the Current Object
- ☐ I can trace self-referencing expressions such as `this.steps + amount` and `this == other`. Reread: 3.9.1 `this` Refers to the Current Object
- ☐ I can use `this` to store a parameter in a shadowed instance variable, and spot code that forgets to. Reread: 3.9.2 Reaching a Hidden Instance Variable
- ☐ I can trace code that passes `this` as an argument and explain why the called method changes the same object. Reread: 3.9.3 Passing `this` as an Argument
- ☐ I can explain why a class method, including `main`, has no `this`, and fix code that uses it there. Reread: 3.9.4 Class Methods Have No `this`

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

105
90

Line	Call	this refers to	this.steps after the call
27	leo.walk(30)	leo's object	50 + 30 = 80
28	mia.walk(5)	mia's object	100 + 5 = 105
29	leo.walk(10)	leo's object	80 + 10 = 90

Each call changes only the object that `this` refers to. `mia`'s object is touched once, so it ends at 105; `leo`'s object is touched twice, so it ends at 90. If you added all three amounts to one total, you treated `this` as one shared object instead of "whichever object the call was made on".

Reread 3.9.1 *this Refers to the Current Object* if your trace differed.

Question 2

B. `this` holds the reference used to make the call, which is the variable before the dot: `back`.

- **A** is wrong: `front` is the argument, so its reference goes into the parameter `other`, not into `this`.
- **C** is wrong: the order of creation never matters; only the call does.
- **D** is wrong: `this` exists in every instance method as well as in constructors.

Reread 3.9.1 *this Refers to the Current Object* if you chose differently.

Question 3

Line 8 is the problem. Inside the constructor, the parameter `name` shadows the instance variable `name`, so `name = name;` copies the parameter into itself. The instance variable keeps its default value, `null`. Line 9 works because `this.level` names the instance variable.

Corrected line 8:

```
1      this.name = name;
```

With that change the program prints `Zara is at level 3`. Java gave no error for the original, because assigning a variable to itself is legal; it just does nothing useful.

Reread 3.9.2 *Reaching a Hidden Instance Variable* if you missed it.

Question 4

B. Line 12 declares a new **local** variable named `target`. Its right side reads `this.target` (68) and adds 4, so the local variable holds 72, and line 13 prints `New target: 72`. But the local variable disappears when the method ends; the instance variable was never assigned, so `getTarget()` returns 68.

- **A** is wrong: it assumes line 12 changes the instance variable. Because of the type `int` at the start, line 12 creates a separate local variable instead. Writing `this.target = this.target + amount;` would change the object.
- **C** is wrong: line 13 prints the plain name `target`, which inside `raise` means the local variable, 72.
- **D** is wrong: this is legal; the local name simply shadows the instance variable (Topic 3.8).

Reread 3.9.2 *Reaching a Hidden Instance Variable* if you chose differently.

Question 5

```
Noor waves at Kai
Kai waves back at Noor
Kai waves at Noor
Noor waves back at Kai
```

Line	Call	this	parameter	Printed
33	<code>noor.greet(kai)</code>	Noor	<code>other = Kai</code>	Noor waves at Kai
18	<code>other.waveBack(this)</code>	Kai	<code>greeter = Noor</code>	Kai waves back at Noor
34	<code>kai.greet(noor)</code>	Kai	<code>other = Noor</code>	Kai waves at Noor
18	<code>other.waveBack(this)</code>	Noor	<code>greeter = Kai</code>	Noor waves back at Kai

Inside `greet`, `this` is the greeter, and it is passed to `waveBack`, where it lands in the parameter `greeter`. Inside `waveBack`, `this` is the friend who was greeted, so `name` there is that friend's name.

Reread 3.9.3 *Passing this as an Argument* if your output differed.

Question 6

Model answer: `this` holds a reference to the current pet, not the pet itself. When a reference is passed as an argument, the parameter `patient` gets a copy of the **reference**, so it refers to the very same object as `rex`. When `examine` calls `patient.markHealthy()`, it changes that one shared object, which is why `rex.isHealthy()` prints `true`.

A complete answer says:

- `this` is a reference to the current object.
- Passing a reference copies the reference, not the object (Topic 3.6).
- So the parameter and `rex` refer to the same object, and changes through the parameter are seen through `rex`.

Reread 3.9.3 *Passing this as an Argument* if your answer was missing one of these.

Question 7

A. `turnOff` is a class method, so there is no current object and no `this`. Java reports non-static variable `this` cannot be referenced from a static context.

- **B** compiles: a class method that uses no instance data needs no object.
- **C** compiles: `dim` is an instance method, so it has a `this`, the lamp it is called on.
- **D** compiles: the class method gets its object through the parameter `lamp`, and it may use the private `brightness` because `lamp` is a `Lamp` and the method is inside `Lamp`.

Reread 3.9.4 *Class Methods Have No this* if you chose differently.

Question 8

An instance method call with no object before the dot, such as `printScore();`, means `this.printScore();`: call it on the current object. That works inside another instance method, but line 18 is in `main`, a class method, which has no `this`. Java cannot tell which object's score to print.

Corrected line 18:

```
1      game.printScore();
```

Now the method runs with `this` referring to the object created on line 17, and the program prints
Score: 40.

Reread 3.9.4 *Class Methods Have No this* if you missed it.

Question 9

B. A constructor runs as part of `new`, and while it runs, the new object is the current object. That is why `this.title = title;` in a constructor fills in the new object.

- **A** is wrong: `this` depends on the call that is running, not on the order of creation.
- **C** is wrong: a `static` method has no current object, so `this` is a compile error there no matter how many objects exist.
- **D** is wrong: assignment copies from right to left, so the parameter `score` is copied into the instance variable `this.score`.

Reread 3.9.1 *this Refers to the Current Object*, 3.9.2 *Reaching a Hidden Instance Variable*, and 3.9.4 *Class Methods Have No this* if you chose differently.

Question 10

```
8
10
```

Line	What happens	red count	blue count
34–35	Two counters are built	7	9
36	<code>red.larger(blue)</code> : <code>this</code> is red, <code>other</code> is blue; <code>7 >= 9</code> is false, so return <code>other</code>	7	9
36	<code>winner</code> now refers to blue's object	7	9
37	<code>winner.increment()</code> changes blue's object	7	10
38	<code>red.larger(red)</code> : <code>this</code> and <code>other</code> are both red; <code>7 >= 7</code> is true, so return <code>this</code>	7	10
39	<code>again</code> refers to red's object; <code>increment</code> changes it	8	10

Returning `this` returns a reference to the current object, not a copy (Topic 3.6), so `winner` and `again` are aliases of existing counters. `other.count` on line 12 is allowed even though `count` is private, because `other` is a `Counter` and the code is inside `Counter`.

Reread 3.9.1 *this Refers to the Current Object* and 3.9.3 *Passing this as an Argument* if your trace differed.

Question 11

`examine` has one parameter of type `Pet`, so its argument must be a `Pet`. The student passed `doctor`, which is a `Vet`, so the types do not match. The pet that should be examined is the current object, the one `visit` was called on, and inside `visit` the way to name it is `this`.

Corrected line 45:

```
1      doctor.examine(this);
```

Inside `Pet.visit`, `this` is a `Pet`, so the call compiles, and the vet examines the pet that made the visit.

Reread 3.9.3 *Passing this as an Argument* if you missed it.

Question 12

A. In `transferTo`, `this` and `other` both refer to Ben's one account. Line 26 subtracts 10 (balance 10). Line 27 calls `receive` on that same account and passes `this`, so in `receive`, `this` and `sender` are also Ben. Line 32 adds 10 back (balance 20), and line 33 prints Ben got 10 from Ben.

- **B** is wrong: it forgets that `receive` runs on the same object and adds the money back.
- **C** is wrong: it treats the receiving account as a separate object that started at 20, but there is only one account.
- **D** is wrong: nothing stops a method from receiving a reference to its own object; this one simply refers to the same object as `this`.

Reread 3.9.3 *Passing this as an Argument* if you chose differently.

Question 13

```
0 members, newest: nobody
2 members, newest: Bea
```

Line	What happens	members	newest
44	The <code>Club</code> constructor sets the starting state	0	"nobody"
45	Prints the first report	0	"nobody"
46	Student constructor: <code>this</code> is the new student; line 30 stores "Amir"; line 31 passes <code>this</code> to <code>join</code>	1	"Amir"
47	Same for the second student, whose name is "Bea"	2	"Bea"
48	Prints the second report	2	"Bea"

Inside `join`, `s` refers to the student under construction, so `s.getName()` returns the name line 30 just stored. Order matters: if line 31 came before line 30, `join` would see the name as `null`.

Reread 3.9.2 *Reaching a Hidden Instance Variable* and 3.9.3 *Passing this as an Argument* if your trace differed.

Question 14

Model answer: `isFrontRow` is a class method, so it is called on the class, as in `Ticket.isFrontRow()`, with no `ticket` in front of the dot. A class method has no current object, so `this` does not exist inside it, and Java reports that `this` cannot be referenced from a static context. One fix is to remove `static`, making it an instance method that is called on a `ticket`, such

as `front.isFrontRow()`; then `this` refers to that ticket. Another fix is to keep it static but add a parameter, `isFrontRow(Ticket t)`, and use `t.seat` instead of `this.seat`.

A complete answer says:

- The method is static, so there is no current object and no `this`.
- Either make it an instance method (remove `static`) or pass the ticket in as a parameter and use the parameter instead of `this`.

Reread *3.9.4 Class Methods Have No this* if your answer was missing one of these.