

1.13 Object Creation and Storage

(Instantiation)

Self-Study Guide · About 60 minutes

By the end you will be able to

- Create an object by writing new followed by a call to one of the class's constructors.
- Declare a variable of the right type to hold a reference to an object, or null when there is no object.
- Read a constructor header and state its signature.
- Explain how constructor parameters give a new object its starting values.
- Decide which overloaded constructor a call uses, and whether a call's arguments match in number, order, and type.
- Trace a constructor call, including copied argument values and the jump into and back out of the constructor.

What to remember from earlier

- A class is a blueprint and an object is one specific thing built from it. ([Topic 1.12: Objects: Instances of Classes, 1.12.1 Classes and Objects](#))
- A reference variable holds a reference to an object, not the object itself. ([Topic 1.12: Objects: Instances of Classes, 1.12.4 Reference Variables](#))
- A reference variable can hold null, meaning no object. ([Topic 1.4: Assignment Statements and Input, 1.4.5 Reference Variables and null](#))
- A signature is the method name plus the ordered parameter types, and same-name methods with different signatures are overloaded. ([Topic 1.9: Method Signatures, 1.9.5 Method Signatures](#))
- Arguments are passed by copying their values (call by value). ([Topic 1.9: Method Signatures, 1.9.6 Call by Value](#))
- Java converts an int to a double automatically, but not the other way. ([Topic 1.5: Casting and Range of Variables, 1.5.1 Java Widens an int to a double](#))

1.13.1 Constructors and the new Keyword

A class is only a blueprint. A blueprint for a house is not a house you can live in; someone has to build the house from it. In the same way, before your program can use a `String` holding your name or a `Scanner` reading the keyboard, an actual object has to be built from the class. How does a program ask Java to build one?

Every class comes with special blocks of code whose only job is to build objects of that class. Think of them as the construction crew that comes with the blueprint. You ask for a new object, the crew builds it, and you get back the object's address.

That construction code is called a **constructor**: a block of code in a class that runs to create a new object of that class. Two facts make constructors easy to spot:

- A constructor has **the same name as its class**, capital letter and all. The `String` class has constructors named `String`; the `Scanner` class has constructors named `Scanner`.
- You call a constructor by writing the keyword `new` in front of it. `new` tells Java "build a brand-new object," and the constructor call after it says which class to build and with what information.

Creating an object this way is called **instantiation**, because the result is a new instance of the class (the word "instance" from Topic 1.12).

```
1 String greeting = new String("Good morning");
```

Piece	What it is
<code>String greeting</code>	Declares a variable named <code>greeting</code> that can hold a reference to a <code>String</code>
<code>=</code>	Stores the result of the right side in the variable (Topic 1.4)
<code>new</code>	Asks Java to build a new object
<code>String(...)</code>	The constructor call: same name as the class, followed by parentheses
<code>"Good morning"</code>	The information passed to the constructor: the characters the new object holds

KEY IDEA

An object is created by writing `new` followed by a call to one of the class's constructors. A constructor always has the same name as its class.

WORKED EXAMPLE Building a String object

Read the two statements and think about what exists in memory after each.

```
1 String greeting = new String("Good morning");
2 System.out.println(greeting);
```

1. Java evaluates the right side of line 1 first. `new` asks for a new object, and the call `String("Good morning")` runs the `String` constructor, which builds a `String` object holding the characters `Good morning`.
2. The result of `new` is a reference to that new object (its address). The `=` stores that reference in `greeting`.
3. Line 2 prints the object that `greeting` refers to.

```
Good morning
```

Now change the class. You have used this line since Topic 1.4, perhaps without knowing what each word meant:

KeyboardScanner.java

```
1 import java.util.Scanner;
2
3 public class KeyboardScanner
4 {
5     public static void main(String[] args)
6     {
7         Scanner keyboard = new Scanner(System.in);
8         System.out.println("A Scanner object is ready.");
9     }
10 }
```

Line 7 follows exactly the same pattern. `Scanner keyboard` declares a variable that can refer to a `Scanner`. `new Scanner(System.in)` calls the `Scanner` constructor, passing it `System.in`, which stands for the keyboard. The constructor builds a `Scanner` object connected to the keyboard, and its reference is stored in `keyboard`. The program then prints its message:

```
A Scanner object is ready.
```

Nothing was typed, and nothing needed to be: creating the `Scanner` only sets it up to read. Reading happens later, when you ask it for input. The class changed (`String` to `Scanner`) and so did the information passed in, but the shape `Type name = new Type(...);` stayed the same.

(You have probably also written `String name = "Ada";` without `new`. A string literal is a shortcut that `String` alone gets; Topic 1.15 covers it. For every other class, objects come from `new`.)

WATCH OUT

Leaving out `new`. Because a constructor call looks like a method call from Topic 1.9, students sometimes write it without `new`:

```
1 String greeting = String("Good morning");
2 System.out.println(greeting);
```

Java reports `cannot find symbol` and names the missing symbol as `method String(String)`. Without `new`, Java thinks you are calling an ordinary method named `String`, and there is no such method. The fix is `String greeting = new String("Good morning");`. The word `new` is what tells Java this is a request to build an object.

PAUSE AND TRY**Question 1** Multiple choice

Which statement creates a new `String` object holding `Hello` by calling a constructor?

- A. `String word = String("Hello");`
- B. `String word = new String("Hello");`
- C. `new String word = "Hello";`
- D. `String word = new("Hello");`

Question 2 Debugging

This program is supposed to create a `Scanner` that reads from the keyboard, but it does not compile. Find the error and write the corrected line.

ScannerNoNew.java

```
1 import java.util.Scanner;
2
3 public class ScannerNoNew
4 {
5     public static void main(String[] args)
6     {
7         Scanner input = Scanner(System.in);
8         System.out.println("Ready");
9     }
10 }
```

1.13.2 Reference Variables and null

Building an object is only half the job. If the reference that `new` hands back is not stored anywhere, your program has no way to find the object again. So where does it go, and what type must that variable be?

Recall from Topic 1.12 that a variable of a reference type holds a **reference**, which you can think of as the memory address of an object, not the object itself. Picture the object as a house and the variable as a sticky note with the house's address written on it.

The variable's type must match the kind of object it will refer to. A `String` variable holds references to `String` objects; a `Scanner` variable holds references to `Scanner` objects. The type is the class name, so declaring the variable looks just like Topic 1.2, with a class name where a primitive type would go: `Scanner keyboard;`, `String city;`.

There is one more thing a reference variable can hold. Sometimes there is no object yet, or no object at all. For that, Java has the literal `null` (from Topic 1.4), a special value meaning "this variable does not refer to any object." On the sticky-note picture, `null` is a note with no address on it. Where the picture stops working: a real sticky note can be blank because someone forgot; `null` is a deliberate value you can assign and print.

KEY IDEA

A variable of a reference type holds either a reference to an object of its type or `null` when there is no object.

WORKED EXAMPLE Two variables, one object

Here one object is created, and then its reference is copied into a second variable. Watch how many objects exist.

```
1 String city = new String("Denver");
2 String sameCity = city;
3 System.out.println(city);
4 System.out.println(sameCity);
```

Line 1 builds one `String` object and stores its reference in `city`. Line 2 has no `new`, so it builds nothing; `=` copies the reference already stored in `city` into `sameCity`. If we call the object's address "address A," memory after line 2 looks like this:

Variable	Holds	Object at that address
<code>city</code>	reference to address A	String "Denver"
<code>sameCity</code>	reference to address A	(the same object)

Both variables lead to the same object, so both lines print the same text:

```
Denver
Denver
```

Two variables, but only **one** object, because `new` ran only once. Count the `new`s to count the objects.

Now vary the starting value. This time the variable starts with no object at all:

```
1 String nickname = null;
2 System.out.println(nickname);
3 nickname = new String("Sky");
4 System.out.println(nickname);
```

After line	nickname holds	Printed
1	<code>null</code>	
2	<code>null</code>	<code>null</code>
3	reference to "Sky"	
4	reference to "Sky"	Sky

When `println` is given a variable holding `null`, it prints the word `null`. Line 3 then creates an object and stores its reference, replacing `null` just as any assignment replaces an old value.

WATCH OUT

Choosing the wrong variable type. A variable can only hold references to its own type. Storing a `Scanner` in a `String` variable does not compile:

WrongVariableType.java

```
1 import java.util.Scanner;
2
3 public class WrongVariableType
4 {
5     public static void main(String[] args)
6     {
7         String keyboard = new Scanner(System.in);
8         System.out.println("Ready");
9     }
10 }
```

Java reports incompatible types: Scanner cannot be converted to String. The fix is to declare the variable with the class that comes after new:

```
Scanner keyboard = new Scanner(System.in);
```

Putting null in a primitive variable. null means "no object," so it only makes sense for reference types:

```
1 int count = null;
2 System.out.println(count);
```

Java reports incompatible types: <null> cannot be converted to int. An int variable always holds a number, never a reference, so it cannot hold null.

PAUSE AND TRY**Question 3** Multiple choice

Which declaration does **not** compile?

- A. String title = null;
- B. Scanner input = new Scanner(System.in);
- C. double price = null;
- D. String title = new String("Up");

Question 4 Tracing

What does this code print? How many String objects does it create?

```
1 String captain = null;
2 String backup = new String("Rosa");
3 System.out.println(captain);
4 captain = backup;
5 System.out.println(captain);
6 System.out.println(backup);
```

1.13.3 Reading a Constructor Signature

So far you have been given the right constructor call. When you meet a new class, though, how do you find out what to put in the parentheses? The same way you found out how to call a method in Topic 1.9: by reading its header in the documentation.

Here are two real constructor headers from the Java documentation, one for each class you know:

```
1 String(String original)
2 Scanner(InputStream source)
```

A constructor header looks a lot like a method header, with two differences. Its name is the class name. And there is **no return type** in front of it, not even `void`, because a constructor's job is fixed: it builds an object of its own class.

The part in parentheses is the **parameter list**. For each value the constructor needs, it lists the value's type and a parameter name: `String original` means "give me a `String`; inside the constructor it will be called `original`." `InputStream source` means "give me an `InputStream`" (the type of `System.in`, the keyboard).

From the header you can read off the constructor's **signature**: the constructor's name followed by the **ordered list of parameter types**. The parameter names are left out of the signature.

Header	Signature
<code>String(String original)</code>	<code>String(String)</code>
<code>Scanner(InputStream source)</code>	<code>Scanner(InputStream)</code>
<code>String()</code>	<code>String()</code>

The last row is a constructor with an empty parameter list: it needs no information at all. Its signature is just the name with empty parentheses.

Why do the parameter names drop out? Because the caller never uses them. When you write `new String("Hi")`, all Java has to check is that you passed one `String`. Whether the constructor calls it `original` or `text` inside makes no difference from outside.

KEY IDEA

A constructor signature is the constructor's name (the same as the class name) plus the ordered list of parameter types. The header also lists parameter names, but the names are not part of the signature.

WORKED EXAMPLE Reading a made-up class's constructor

`String` and `Scanner` hide their constructor code. To see inside a constructor, this chapter uses a small class invented for it, `Ticket`, which stands for a movie ticket. You will write classes and constructors yourself in Unit 3; here you only read them. Its constructor header is:

```
1 Ticket(String movie, int seat, double price)
```

Take it apart one piece at a time:

Piece of the header	Meaning
Ticket	Same name as the class, so this is a constructor of Ticket
String movie	First value needed: a String, called movie inside
int seat	Second value needed: an int, called seat inside
double price	Third value needed: a double, called price inside

Drop the names and keep the types in order: the signature is `Ticket(String, int, double)`. To call it, you need three values, in that order: a `String`, then an `int`, then a `double`.

Order is part of the signature. A constructor with header

`Ticket(int seat, String movie, double price)` would have the signature

`Ticket(int, String, double)`, a different signature, even though it uses the same three types.

WATCH OUT

Treating the parameter names as part of the signature. Headers

`Ticket(String movie, int seat, double price)` and

`Ticket(String title, int row, double cost)` look different, but drop the names and both are `Ticket(String, int, double)`: the same signature. A class cannot have two constructors with the same signature, so putting both in one class does not compile:

DuplicateSignature.java

```
1 public class DuplicateSignature
2 {
3     public static void main(String[] args)
4     {
5         Ticket myTicket = new Ticket("Dune", 14, 12.5);
6     }
7 }
8
9 class Ticket
10 {
11     Ticket(String movie, int seat, double price)
12     {
13         System.out.println(movie);
14     }
15
16     Ticket(String title, int row, double cost)
17     {
18         System.out.println(title);
19     }
20 }
```

Java reports constructor `Ticket(String,int,double)` is already defined in class `Ticket`. Notice that the compiler's own description of the constructor is the signature, with no names in it.

PAUSE AND TRY**Question 5** Short answer

The `Scanner` class also has a constructor with the header `Scanner(String source)`. What is its signature? How many arguments does a call to it need, and of what type?

Question 6 Multiple choice

Which of these is **not** part of a constructor's signature?

- A. The constructor's name
- B. The types of its parameters
- C. The order of its parameters
- D. The names of its parameters

1.13.4 Parameters Set Starting Values

Why do constructors take parameters at all? Because a brand-new object has to start out holding something. A `String` object holds characters, but which characters? A `Scanner` reads text, but from where? The object cannot guess; the code that asks for it has to say.

That is the job of constructor parameters. When you call a constructor, the values you pass in become the object's starting values: the initial values of its attributes (the data each object keeps, from Topic 1.12). Different values passed in mean different objects come out.

Here is what that means for the `String(String original)` constructor. Its documentation says the new object holds the same characters as the `String` it was given. So the argument decides the new object's characters.

KEY IDEA

Parameters let a constructor receive values that set the starting values of the new object's attributes.

WORKED EXAMPLE Same class, different starting values

Both objects come from the same constructor. Only the value passed in changes.

```
1 String first = new String("Ada");
2 String second = new String("Grace");
3 System.out.println(first);
4 System.out.println(second);
```

Line	Object built	Its starting characters	Reference stored in
1	<code>new String</code>	Ada	<code>first</code>
2	<code>new String</code>	Grace	<code>second</code>

Lines 3 and 4 print each object's characters:

```
Ada
Grace
```

Same blueprint, same constructor, two different objects, because each call passed a different starting value.

Now vary the class. `Scanner` has a constructor `Scanner(String source)` whose parameter sets the text the `Scanner` will read from, instead of the keyboard. That makes a handy, predictable example, because no one has to type anything:

ScannerFromText.java

```
1 import java.util.Scanner;
2
3 public class ScannerFromText
4 {
5     public static void main(String[] args)
6     {
7         Scanner reader = new Scanner("85 92");
8         int firstScore = reader.nextInt();
9         int secondScore = reader.nextInt();
10        System.out.println(firstScore);
11        System.out.println(secondScore);
12    }
13 }
```

Line 7 builds a `Scanner` whose starting text is `85 92`. Line 8 calls `nextInt`, which (as in Topic 1.4) reads the next whole number, so `firstScore` gets 85; line 9 reads the next one, so `secondScore` gets 92. (Calling methods on an object, like `reader.nextInt()`, is taught fully in Topic 1.14.)

```
85
92
```

The argument `"85 92"` decided what this `Scanner` would read. Pass `"60 75"` instead, and the same lines would print 60 and 75.

WATCH OUT

Thinking an object with no starting characters is the same as `null`. The constructor `String()` takes no arguments and builds a `String` object with no characters in it. That is still an object; `null` means there is no object at all.

```
1 String empty = new String();
2 String missing = null;
3 System.out.print("[");
4 System.out.print(empty);
5 System.out.println("]");
6 System.out.print("[");
7 System.out.print(missing);
8 System.out.println("]");
```

```
[]
[null]
```

The empty object prints nothing between the brackets. The `null` variable prints the word `null`, because it refers to no object. Students mix these up because both seem to mean "nothing," but one is an empty object that exists and the other is the absence of an object.

PAUSE AND TRY

Question 7 Tracing

What does this program print?

MinutesReader.java

```
1 import java.util.Scanner;
2
3 public class MinutesReader
4 {
5     public static void main(String[] args)
6     {
7         Scanner times = new Scanner("40 15");
8         String runner = new String("Jo");
9         int warmUp = times.nextInt();
10        int coolDown = times.nextInt();
11        System.out.println(runner);
12        System.out.println(coolDown);
13        System.out.println(warmUp);
14    }
15 }
```

Question 8 Short answer

After `String a = new String();` and `String b = null;`, how many `String` objects exist? Explain what each variable holds.

1.13.5 Overloaded Constructors

You have now seen two `String` constructors, `String()` and `String(String original)`, and two `Scanner` constructors, one taking `System.in` and one taking a `String`. Why would a class have more than one way to build an object?

Because people want objects that start out in different ways. Sometimes you want a `String` with given characters; sometimes you want an empty one. Sometimes a `Scanner` should read the keyboard; sometimes it should read text you already have. Rather than invent a different name for each, the class offers several constructors with the same name, each with its own parameter list. This is the same idea as overloaded methods in Topic 1.9.

Constructors are **overloaded** when a class has more than one constructor with different signatures. Since every constructor of a class has the same name, "different signatures" means the parameter lists differ: a different number of parameters, different types, or the same types in a different order.

How does Java know which one you mean? It compares your arguments to each signature and picks the one whose parameter list matches: same number, and compatible types in the same order. That is exactly why two constructors in one class may never share a signature: Java would have no way to choose.

KEY IDEA

Overloaded constructors share the class name but have different signatures. Java chooses which one runs by matching the call's arguments to each parameter list.

WORKED EXAMPLE Which Scanner constructor runs?

This program builds two `Scanner` objects with two different constructors.

TwoScanners.java

```

1  import java.util.Scanner;
2
3  public class TwoScanners
4  {
5      public static void main(String[] args)
6      {
7          Scanner keyboard = new Scanner(System.in);
8          Scanner scores = new Scanner("70 88");
9          int score = scores.nextInt();
10         System.out.println(score);
11     }
12 }

```

Match each call to a signature:

Line	Call	Argument type	Constructor that runs
7	<code>new Scanner(System.in)</code>	InputStream	Scanner(InputStream)
8	<code>new Scanner("70 88")</code>	String	Scanner(String)

Both calls pass one argument, so the number alone cannot decide. The argument's **type** decides. Line 9 reads the first number from the text-based Scanner, so the program prints:

```
70
```

Now vary the number of arguments instead of the type. For String:

Call	Number of arguments	Constructor that runs	New object holds
<code>new String()</code>	0	String()	no characters
<code>new String("Lake")</code>	1 (String)	String(String)	Lake

Here the count of arguments is enough to tell the two constructors apart.

WATCH OUT

Assuming there is a constructor for whatever you pass. Overloading gives a class several constructors, but only the ones actually written. Hoping to turn a number into text, a student writes:

```

1  String digits = new String(5);
2  System.out.println(digits);

```

Java reports no suitable constructor found for `String(int)`, then lists the `String` constructors it tried (including some this course never uses) and why each failed. None of them takes an `int`. Before calling a constructor, check that a signature matching your arguments really exists in the documentation.

PAUSE AND TRY**Question 9** Multiple choice

An invented class `Lamp` has three constructors:

```
1 Lamp()  
2 Lamp(int brightness)  
3 Lamp(String color, int brightness)
```

Which constructor does `new Lamp(75)` call?

- A. `Lamp()`
- B. `Lamp(int brightness)`
- C. `Lamp(String color, int brightness)`
- D. None; the call does not compile because there are three `Lamp` constructors.

Question 10 Short answer

Using the `Lamp` constructors from the previous question, which constructor does `new Lamp("red", 40)` call? Explain why `new Lamp(40, "red")` would not compile.

1.13.6 Arguments Must Match the Parameter List

A signature tells you what a constructor needs. What exactly counts as giving it what it needs, and what happens if you get it slightly wrong?

As in Topic 1.9, an **argument** is a value passed in when the call is made: the values between the parentheses after `new Ticket`. The parameters are the variables in the header that receive them. Java pairs them up by position: the first argument goes to the first parameter, the second to the second, and so on. That pairing works only if three things line up:

- **Number:** exactly as many arguments as parameters.
- **Order:** the arguments come in the same order as the parameter types.
- **Type:** each argument is **compatible** with its parameter's type, meaning it is that type or one Java converts to it automatically. From Topic 1.5, an `int` converts to a `double` automatically; a `double` never converts to an `int` on its own, because that could lose the decimal part.

Why so strict? Java pairs by position, not by meaning. It cannot tell that `14` "looks like" a seat number; it only knows the second parameter is an `int`.

KEY IDEA

The arguments in a constructor call must match the parameter list in number and order, and each argument must be compatible with its parameter's type.

WORKED EXAMPLE A call that matches

This version of `Ticket`'s constructor prints the three values it receives, so you can see the pairing.

`TicketArguments.java`

```

1  public class TicketArguments
2  {
3      public static void main(String[] args)
4      {
5          Ticket myTicket = new Ticket("Dune", 14, 12);
6      }
7  }
8
9  class Ticket
10 {
11     Ticket(String movie, int seat, double price)
12     {
13         System.out.println(movie);
14         System.out.println(seat);
15         System.out.println(price);
16     }
17 }
```

Pair each argument on line 5 with the header on line 11:

Position	Argument	Parameter	Value the parameter gets
1	"Dune"	String movie	"Dune"
2	14	int seat	14
3	12	double price	12.0 (int converted to double)

Three arguments for three parameters, in the right order. The third argument is an `int` but the parameter is a `double`, which is compatible, so Java converts it. The output shows the conversion:

```

Dune
14
12.0
```

Now vary the order and the count. Each of these calls is one small change from line 5 above, and none of them compiles:

Call	What is wrong	Gist of the compiler message
<code>new Ticket(14, "Dune", 12.5)</code>	First two arguments swapped (order)	incompatible types: int cannot be converted to String
<code>new Ticket("Dune", 14)</code>	Missing the price (number)	actual and formal argument lists differ in length
<code>new Ticket("Dune", 12.5, 14)</code>	12.5 sent to int seat (type)	incompatible types: possible lossy conversion from double to int

The type mistake is the easiest to miss, so here it is as a full program:

TicketLossy.java

```
1 public class TicketLossy
2 {
3     public static void main(String[] args)
4     {
5         Ticket myTicket = new Ticket("Dune", 12.5, 14);
6     }
7 }
8
9 class Ticket
10 {
11     Ticket(String movie, int seat, double price)
12     {
13         System.out.println(movie);
14     }
15 }
```

The third argument 14 going to double price would be fine. The problem is the second: 12.5 cannot go into an int without losing .5, so Java refuses.

WATCH OUT

Ordering arguments by what makes sense instead of by the header. Students often write the values in the order they think of them ("seat 14 for Dune") rather than the order of the parameter list. Java matches by position only, so always check the header and line up each argument with its parameter, left to right.

PAUSE AND TRY

Question 11 Multiple choice

Ticket has only the constructor `Ticket(String movie, int seat, double price)`. Which call compiles?

- A. `new Ticket("Up", 7.5, 3)`
- B. `new Ticket("Up", 3, 7)`
- C. `new Ticket("Up", 3)`
- D. `new Ticket(3, "Up", 7.5)`

Question 12 Debugging

This program should create a ticket for seat 4 of *Up* costing 8.5, but it does not compile. Explain the error and write the corrected line.

TicketMissingPrice.java

```
1 public class TicketMissingPrice
2 {
3     public static void main(String[] args)
4     {
5         Ticket matinee = new Ticket("Up", 4);
6     }
7 }
8
9 class Ticket
10 {
11     Ticket(String movie, int seat, double price)
12     {
13         System.out.println(movie);
14     }
15 }
```

1.13.7 Call by Value: Parameters Get Copies

When you hand a value to a constructor and the constructor changes its parameter, does your own variable change too? It matters: if passing a variable could change it behind your back, you could never trust a variable's value after a call.

Topic 1.9 answered this for methods, and constructors work the same way. Arguments are passed using **call by value**: each parameter starts out holding a **copy** of its argument's value. The constructor works with its copy. Your variable and the parameter are two separate boxes that happen to start with the same value.

Picture it like photocopying a form before handing it over. Whatever the other person writes on the copy, your original stays as it was.

KEY IDEA

With call by value, a constructor's parameters are initialized with copies of the arguments. Changing a parameter inside the constructor does not change the caller's variable.

WORKED EXAMPLE Changing the copy, not the original

In this version, the constructor adds 100 to its `seat` parameter. Watch `mySeat` in `main`.

SeatCopy.java

```
1 public class SeatCopy
2 {
3     public static void main(String[] args)
4     {
5         int mySeat = 14;
6         Ticket myTicket = new Ticket("Dune", mySeat, 12.5);
7         System.out.print("Back in main, mySeat is ");
8         System.out.println(mySeat);
9     }
10 }
11
12 class Ticket
13 {
14     Ticket(String movie, int seat, double price)
15     {
16         seat = seat + 100;
17         System.out.print("Inside the constructor, seat is ");
18         System.out.println(seat);
19     }
20 }
```

Step	mySeat (in main)	seat (in constructor)	Printed
Line 5	14	(does not exist yet)	
Line 6: call starts	14	14 (a copy)	
Line 16	14	114	
Lines 17–18	14	114	Inside the constructor, seat is 114
Back in main, lines 7–8	14	(gone)	Back in main, mySeat is 14

```
Inside the constructor, seat is 114
Back in main, mySeat is 14
```

Only the copy changed. `mySeat` was never touched.

Now vary the argument: it does not have to be a variable. It can be any expression of a compatible type. Java evaluates the expression first, then passes a copy of the result.

SeatExpression.java

```
1 public class SeatExpression
2 {
3     public static void main(String[] args)
4     {
5         int row = 3;
6         Ticket myTicket = new Ticket("Dune", row * 10, 12.5);
7         System.out.print("row is ");
8         System.out.println(row);
9     }
10 }
11
12 class Ticket
13 {
14     Ticket(String movie, int seat, double price)
15     {
16         System.out.print("seat is ");
17         System.out.println(seat);
18     }
19 }
```

On line 6, `row * 10` is evaluated to 30 before the constructor runs, and `seat` starts as a copy of 30. Computing the argument does not change `row`, so the program prints:

```
seat is 30
row is 3
```

WATCH OUT

Thinking the parameter is the same variable as the argument. Students expect `mySeat` to become 114 because "it was passed in." But the constructor never sees `mySeat`, only a copy of its value. The same holds even if the names match: if `main`'s variable were also called `seat`, it would still be a separate variable from the parameter `seat`.

PAUSE AND TRY**Question 13** Tracing

What does this program print?

PriceCopy.java

```
1 public class PriceCopy
2 {
3     public static void main(String[] args)
4     {
5         double cost = 10.0;
6         Ticket myTicket = new Ticket("Up", 8, cost);
7         System.out.println(cost);
8     }
9 }
10
11 class Ticket
12 {
13     Ticket(String movie, int seat, double price)
14     {
15         price = price / 2;
16         System.out.println(price);
17     }
18 }
```

1.13.8 Constructor Calls and the Flow of Control

In the last example, the constructor's line printed **before** `main`'s next line, even though it sits lower in the file. Why? Because of the order in which the statements actually run.

Back in Topic 1.1, statements ran one after another, top to bottom. Topic 1.9 added an exception: a method call jumps into the method, runs it, and comes back. A constructor call does the same thing. When Java reaches `new Ticket(...)`, it pauses the current statement, runs every statement in the constructor in order, and once the last one finishes, control returns to the point right after the call: it finishes the paused statement (storing the reference) and moves on to the next line. The order in which statements run is called the **flow of control**.

It is like pausing a video to answer the door. You deal with the visitor completely, then resume the video exactly where you stopped, not from the beginning.

KEY IDEA

A constructor call interrupts the normal top-to-bottom order: the constructor's statements all run first, then control returns to the point just after the call.

WORKED EXAMPLE Following the jump

The letters in the messages show the order in which they print. Predict the output before reading on.

ConstructorFlow.java

```
1 public class ConstructorFlow
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("A: before the constructor call");
6         Ticket myTicket = new Ticket("Dune", 14, 12.5);
7         System.out.println("D: after the constructor call");
8     }
9 }
10
11 class Ticket
12 {
13     Ticket(String movie, int seat, double price)
14     {
15         System.out.println("B: constructor starts");
16         System.out.println("C: constructor ends");
17     }
18 }
```

Order	Line running	Where	Printed
1	5	main	A: before the constructor call
2	6	main: call	(jump into the constructor)
3	15	constructor	B: constructor starts
4	16	constructor	C: constructor ends
5	6	main: resumes	(reference stored in myTicket)
6	7	main	D: after the constructor call

A: before the constructor call
B: constructor starts
C: constructor ends
D: after the constructor call

Now vary the number of calls. Each `new` makes its own jump, and each jump returns to just after its own call:

TwoTickets.java

```
1 public class TwoTickets
2 {
3     public static void main(String[] args)
4     {
5         Ticket first = new Ticket("Dune", 14, 12.5);
6         System.out.println("Between tickets");
7         Ticket second = new Ticket("Up", 3, 9.0);
8         System.out.println("Done");
9     }
10 }
11
12 class Ticket
13 {
14     Ticket(String movie, int seat, double price)
15     {
16         System.out.print("Printing ticket for ");
17         System.out.println(movie);
18     }
19 }
```

Line 5 jumps into the constructor (printing the ticket for *Dune*), comes back, and line 6 runs. Line 7 jumps in again with different arguments, comes back, and line 8 runs.

```
Printing ticket for Dune
Between tickets
Printing ticket for Up
Done
```

WATCH OUT

Reading the file top to bottom as the run order. The constructor's code sits below `main` in the file, so students sometimes predict that its lines print last, or not at all. Code in a constructor runs only when a call reaches it, and it runs right then. Trace by following the calls, not the line numbers on the page.

PAUSE AND TRY**Question 14** Tracing

What does this program print?

MovieNight.java

```
1 public class MovieNight
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("Line up");
6         Ticket mine = new Ticket("Up", 5, 9.0);
7         System.out.println("Buy popcorn");
8         Ticket yours = new Ticket("Up", 6, 9.0);
9         System.out.println("Find seats");
10    }
11 }
12
13 class Ticket
14 {
15     Ticket(String movie, int seat, double price)
16     {
17         System.out.print("Ticket ");
18         System.out.println(seat);
19     }
20 }
```

Putting it together

A box-office program reads a seat number and price from a `Scanner` and books a ticket. It uses every idea from this chapter. (A text-based `Scanner` stands in for the keyboard so the output is always the same.)

BoxOffice.java

```
1 import java.util.Scanner;
2
3 public class BoxOffice
4 {
5     public static void main(String[] args)
6     {
7         Scanner order = new Scanner("14 12");
8         String movie = new String("Dune");
9         int seatNumber = order.nextInt();
10        int dollars = order.nextInt();
11        Ticket booked = null;
12        System.out.println("Booking...");
```

```

13         booked = new Ticket(movie, seatNumber, dollars);
14         System.out.print("Done. seatNumber is still ");
15         System.out.println(seatNumber);
16     }
17 }
18
19 class Ticket
20 {
21     Ticket(String movie, int seat, double price)
22     {
23         System.out.print("Ticket for ");
24         System.out.println(movie);
25         seat = seat + 1;
26         System.out.print("Seat ");
27         System.out.println(seat);
28         System.out.print("Price ");
29         System.out.println(price);
30     }
31 }

```

Order	Line(s)	What happens	Printed
1	7	new calls Scanner(String) (overloading: the argument is a String); order refers to a Scanner reading 14 12	
2	8	new calls String(String); movie refers to a String holding Dune	
3	9–10	seatNumber is 14, dollars is 12	
4	11	booked is declared with type Ticket and holds null: no ticket yet	
5	12	prints	Booking...
6	13	Arguments match Ticket(String, int, double) in number and order; dollars (an int) is compatible with double	(jump into the constructor)
7	23–24	movie parameter holds a copy of the reference to "Dune"	Ticket for Dune
8	25–27	seat starts as a copy of 14 and becomes 15	Seat 15
9	28–29	price is 12.0	Price 12.0
10	13	Control returns; the new object's reference replaces null in booked	
11	14–15	seatNumber is unchanged by call by value	Done. seatNumber is still 14

```

Booking...
Ticket for Dune
Seat 15
Price 12.0
Done. seatNumber is still 14

```

Three objects were created, one per new: a Scanner, a String, and a Ticket.

Vocabulary

Term	Definition	Example
constructor	Code in a class, with the same name as the class, that runs to create a new object	<code>String(String original)</code>
new	The keyword placed before a constructor call to create a new object	<code>new Scanner(System.in)</code>
instantiation	Creating an object (an instance) of a class	<code>new String("Hi")</code>
reference	What a reference-type variable holds: in effect, the address of an object	<code>city</code> holds the address of "Denver"
null	A special value meaning a reference variable refers to no object	<code>String nickname = null;</code>
parameter list	The types and names, in parentheses in a header, of the values a constructor needs	<code>(String movie, int seat, double price)</code>
signature	The constructor's name plus the ordered list of parameter types	<code>Ticket(String, int, double)</code>
overloaded	Describes constructors of one class that share a name but have different signatures	<code>String()</code> and <code>String(String)</code>
argument	A value passed to a constructor when it is called	"Dune" in <code>new Ticket("Dune", 14, 12.5)</code>
compatible	An argument that is the parameter's type or converts to it automatically	12 passed to <code>double price</code>
call by value	Passing arguments by giving each parameter a copy of the argument's value	Changing <code>seat</code> leaves <code>mySeat</code> alone
flow of control	The order in which statements actually run	Constructor lines run before the next line of <code>main</code>

Chapter summary

- An object is created by writing `new` followed by a call to one of the class's **constructors**, which always have the same name as the class. The result is a reference to the new object.
- A reference variable must be declared with the object's class as its type. It holds a **reference** to an object of that type, or `null` when there is no object.
- A constructor's **signature** is its name plus the ordered list of parameter types; parameter names are not part of it. A class can have several **overloaded** constructors with different signatures, and Java picks the one whose parameter list matches the arguments.
- Constructor parameters receive values that set the new object's starting values. The **arguments** must match the parameter list in number and order and be **compatible** in type.
- Arguments are passed **call by value**: parameters get copies, so changing a parameter never changes the caller's variable. A constructor call jumps into the constructor, runs all its statements, then returns to just after the call.

Review questions

Question 15 Multiple choice

A program has `import java.util.Scanner;` at the top. Which statement correctly creates a `Scanner` that reads from the keyboard and stores it so the program can use it later?

- A. `Scanner keyboard = Scanner(System.in);`
- B. `String keyboard = new Scanner(System.in);`
- C. `Scanner keyboard = new Scanner(System.in);`
- D. `new Scanner(System.in);`

Question 16 Short answer

Kai writes `Ticket saved = null;` and says, "This creates a `Ticket` object that has no information in it yet." Is Kai right? Explain, using what `null` means and what has to happen for an object to be created.

Question 17 Multiple choice

An invented class `Song` already has the constructor `Song(String title, int seconds)`. Which constructor could **not** be added to the same class?

- A. `Song(String title)`
- B. `Song(int seconds, String title)`
- C. `Song(String name, int length)`
- D. `Song(String title, int seconds, String artist)`

Question 18 Short answer

`Ticket` has only the constructor `Ticket(String movie, int seat, double price)`. Explain why `new Ticket("Up", 9, 6)` compiles but `new Ticket("Up", 9.0, 6)` does not, even though both use the same three numbers.

Question 19 Debugging

This program should read the first number from the text `5 9` and print it. Line 7 has **two** errors. Find both and write the corrected line.

ReaderMistakes.java

```
1 import java.util.Scanner;
2
3 public class ReaderMistakes
4 {
5     public static void main(String[] args)
6     {
7         String reader = Scanner("5 9");
8         int first = reader.nextInt();
9         System.out.println(first);
10    }
11 }
```

Question 20 Tracing

What does this program print? Track `seat` in `main` separately from the parameter `seat`.

Matinee.java

```
1 public class Matinee
2 {
3     public static void main(String[] args)
4     {
5         int seat = 20;
6         System.out.println("Start");
7         Ticket early = new Ticket("Up", seat, 7.0);
8         System.out.println(seat);
9         Ticket late = new Ticket("Dune", seat + 1, 7.0);
10        System.out.println(seat);
11    }
12 }
13
14 class Ticket
15 {
16     Ticket(String movie, int seat, double price)
17     {
18         seat = seat * 2;
19         System.out.print(movie);
20         System.out.print(" ");
21         System.out.println(seat);
22     }
23 }
```

Reflection

- ☐ I can create an object using `new` and a constructor, and say why the constructor has the class name. Reread: 1.13.1 Constructors and the `new` Keyword
- ☐ I can declare a variable of the right type for an object, and explain what `null` means. Reread: 1.13.2 Reference Variables and `null`
- ☐ I can write the signature of a constructor from its header. Reread: 1.13.3 Reading a Constructor Signature
- ☐ I can explain how constructor arguments set a new object's starting values. Reread: 1.13.4 Parameters Set Starting Values
- ☐ I can tell which overloaded constructor a call will use. Reread: 1.13.5 Overloaded Constructors
- ☐ I can check whether a call's arguments match in number, order, and type. Reread: 1.13.6 Arguments Must Match the Parameter List
- ☐ I can explain why changing a parameter does not change the caller's variable. Reread: 1.13.7 Call by Value: Parameters Get Copies
- ☐ I can trace the order of output when a constructor call jumps into the constructor and back. Reread: 1.13.8 Constructor Calls and the Flow of Control