

4.7 Wrapper Classes

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

By the end you will be able to

- Explain what an Integer or Double object is and how it differs from an int or double value.
- Use Integer.MIN_VALUE and Integer.MAX_VALUE as safe starting values when finding a maximum or minimum.
- Predict where Java autoboxes a primitive into a wrapper object and where it unboxes a wrapper object into a primitive.
- Explain why a wrapper object can never be changed, and what that means for aliases and method parameters.
- Convert text such as a piece of a file line into a number with Integer.parseInt and Double.parseDouble.

What to remember from earlier

- A primitive variable holds the value itself; a class-type variable holds a reference or null. ([Topic 1.2: Variables and Data Types](#), [1.2.2 Primitive Types and Reference Types](#))
- Integer.MAX_VALUE and Integer.MIN_VALUE are the largest and smallest int. ([Topic 1.5: Casting and Range of Variables](#), [1.5.5 The Range of int Values](#))
- Calling a method through a null reference throws a NullPointerException. ([Topic 1.14: Calling Instance Methods](#), [1.14.5 Calling a Method on null](#))
- A String is immutable. ([Topic 1.15: String Manipulation](#), [1.15.7 Strings Are Immutable](#))
- A method can change a mutable object it receives, but assigning to the parameter never changes the caller's variable. ([Topic 3.6: Methods: Passing and Returning References of an Object](#), [3.6.2 What a Method Cannot Change](#))

4.7.1 The Integer and Double Classes

Java has two kinds of types. Primitive types such as `int` and `double` are simple values: a variable of type `int` holds the number itself. Class types such as `String`, `Scanner`, and the classes you wrote in Unit 3 describe objects, and a variable of a class type holds a reference to an object. Most of the time numbers are happiest as primitives. But some parts of Java are built to hold **objects only**. The next topic's `ArrayList`, a list that can grow, is one of them, and it cannot hold a plain `int`. So Java needs a way to package a number as an object.

The package is a **wrapper class**: a class whose objects each hold one primitive value, "wrapped" inside an object. Picture a number sealed inside a labeled envelope. The envelope can go anywhere an object is required, and the number inside is still there to be read. For the two primitive types this course uses most, the wrapper classes are:

Primitive type	Wrapper class	A variable of the wrapper type holds...
<code>int</code>	<code>Integer</code>	a reference to an object containing one <code>int</code>
<code>double</code>	<code>Double</code>	a reference to an object containing one <code>double</code>

Notice the capital letters. `int` and `double` are primitive types and are lowercase. `Integer` and `Double` are class names, and class names start with a capital letter. Here is the difference in memory:

```
int laps = 12;           laps    [ 12 ]

Integer lapCount = 12;   lapCount [ ref ] → Integer object
                                   value: 12
```

Because `Integer` and `Double` are class types, a variable of those types can also hold `null` (Topic 1.13), meaning it refers to no object. An `int` variable can never be `null`.

Both classes belong to the `java.lang` **package**, the group of classes that Java makes available to every program automatically. `String` and `Math` are in `java.lang` too, which is why you have never imported them. Compare that with `File` from Topic 4.6, which lives in `java.io` and needs an `import` statement.

KEY IDEA

`Integer` and `Double` are wrapper classes: each object holds one `int` or one `double` value. They are in the `java.lang` package, so any program can use them without an `import`.

The Integer constants

You have already used the `Integer` class without knowing much about it. In Topic 1.5 you met `Integer.MAX_VALUE` (2,147,483,647, the largest `int`) and `Integer.MIN_VALUE` (−2,147,483,648, the smallest `int`). Now you can read those names fully: they are public `static` constants that belong to the `Integer` class, so you reach them through the class name, just like the class variables in Topic 3.7.

They are especially useful as starting values for the minimum and maximum algorithms from Topic 4.5. To find the smallest value, you need a starting "smallest so far" that any real value will beat. No `int` is larger than `Integer.MAX_VALUE`, so the first element is always less than or equal to it.

WORKED EXAMPLE The fastest lap

A runner's lap times in seconds are stored in an array. Watch how `fastest` changes as the loop visits each time.

```
1 int[] lapTimes = {74, 69, 81, 72};
2 int fastest = Integer.MAX_VALUE;
3 for (int time : lapTimes)
4 {
5     if (time < fastest)
6     {
7         fastest = time;
8     }
9 }
10 System.out.println(fastest);
```

Visit	time	time < fastest?	fastest afterward
start			2147483647

1	74	yes	74
2	69	yes	69
3	81	no	69
4	72	no	69

The first comparison is guaranteed to be true, so `fastest` takes on a real lap time right away. After that, only smaller times replace it. Line 10 prints:

69

Now vary the goal: find the **largest** value in an array of overnight low temperatures, all below zero. The starting value flips to `Integer.MIN_VALUE`, because no `int` is smaller, so the first element always beats it.

```
1 int[] lows = {-12, -5, -9};
2 int warmest = Integer.MIN_VALUE;
3 for (int low : lows)
4 {
5     if (low > warmest)
6     {
7         warmest = low;
8     }
9 }
10 System.out.println(warmest);
```

-5

`warmest` starts at `-2,147,483,648`. The first low, `-12`, is greater, so it replaces it. Then `-5` is greater than `-12`, and `-9` is not greater than `-5`. The warmest night was `-5`.

WATCH OUT

Starting the maximum at 0. Zero feels like a natural "nothing yet" value, so students often write this:

```
1 int[] lows = {-12, -5, -9};
2 int warmest = 0;
3 for (int low : lows)
4 {
5     if (low > warmest)
6     {
7         warmest = low;
8     }
9 }
10 System.out.println(warmest);
```

0

Every low is negative, so no element is ever greater than 0, and `warmest` never changes. The program prints 0, a temperature that is not even in the array. This is a logic error: it compiles and runs, but the answer is wrong. Starting at 0 only works if you know every value is positive. Starting at `Integer.MIN_VALUE` (or at the first element of the array) works for any `int` values.

PAUSE AND TRY

Question 1 Tracing

Write the exact output.

```
1 int[] jumps = {-3, -8, -1, -6};
2 int biggest = Integer.MIN_VALUE;
3 for (int jump : jumps)
4 {
5     if (jump > biggest)
6     {
7         biggest = jump;
8         System.out.println(biggest);
9     }
10 }
```

Question 2 Short answer

A program that reads a text file needs `import java.io.File;` at the top, but it can use `Integer.MAX_VALUE` with no import at all. Explain why.

4.7.2 Autoboxing

You know what an `Integer` object is. How do you make one? You might expect to call a constructor with `new`, as with every other object so far. You do not need to. If you write `Integer lapCount = 12;`, the compiler sees an `int` value going into an `Integer` variable and quietly adds the code that creates an `Integer` object holding 12. You write a number, and Java seals it in the envelope for you.

This automatic conversion from a primitive value to an object of its wrapper class is called **autoboxing** (the value is put "in a box"). It converts an `int` to an `Integer` and a `double` to a `Double`. The compiler applies it in two situations:

1. A primitive value is **assigned** to a variable of the matching wrapper type.
2. A primitive value is **passed as an argument** to a method whose parameter has the matching wrapper type.

KEY IDEA

Autoboxing is the compiler automatically turning an `int` into an `Integer`, or a `double` into a `Double`, when the value is assigned to a wrapper variable or passed to a wrapper parameter.

WORKED EXAMPLE Boxing by assignment

Lines 2 and 3 store primitive values in wrapper variables.

```
1 int laps = 12;
2 Integer lapCount = laps;
3 Double price = 4.75;
4 System.out.println(lapCount);
5 System.out.println(price);
```

- Line 1 stores the primitive 12 directly in `laps`.
- Line 2 assigns an `int` to an `Integer` variable. The compiler autoboxes: it creates an `Integer` object holding 12, and `lapCount` holds a reference to it. `laps` is unchanged; it still holds its own 12.
- Line 3 does the same with a `double` literal: a `Double` object holding 4.75 is created, and `price` refers to it.
- Lines 4 and 5 print the wrapper variables. Printing a wrapper object (or joining it to a string with `+`) shows the number inside, not an address.

```
12
4.75
```

The second situation is a method call. Here two methods have wrapper parameters. Watch lines 6, 7, and 8.

AutoboxParameter.java

```
1 public class AutoboxParameter
2 {
3     public static void main(String[] args)
4     {
5         int adaPoints = 15;
6         printBadge("Ada", adaPoints);
7         printBadge("Grace", 9);
8         printAverage("Ada", 87.5);
9     }
10
11     public static void printBadge(String name, Integer points)
12     {
13         System.out.println(name + " has " + points + " points");
14     }
15
16     public static void printAverage(String name, Double average)
17     {
18         System.out.println(name + " averages " + average);
19     }
20 }
```

```
Ada has 15 points
Grace has 9 points
Ada averages 87.5
```

On line 6, the argument `adaPoints` is an `int`, but the parameter `points` on line 11 is an `Integer`. The compiler autoboxes the value 15 into a new `Integer` object, and `points` starts out referring to it. Line 7 shows that the argument can be a literal: 9 is boxed the same way. Line 8 boxes the `double` 87.5 into a `Double` for the parameter `average`. Inside the methods, line 13 and line 18 join the wrapper values into a string, which shows their numbers.

You may see older code that builds a wrapper with `new Integer(12)`. That constructor is deprecated (marked for removal) in current Java and is not used on the AP exam; let autoboxing create wrapper objects instead.

WATCH OUT

Boxing into the wrong wrapper. In Topic 1.5 you learned that Java happily widens an `int` to a `double`. So students expect an `int` to fit into a `Double` too:

BoxWrongType.java

```
1 public class BoxWrongType
2 {
3     public static void main(String[] args)
4     {
5         Double price = 5;
6         printPrice("pencil", 2);
7     }
8
9     public static void printPrice(String item, Double price)
10    {
11        System.out.println(item + " costs " + price);
12    }
13 }
```

This does not compile. Java reports (wording can vary between versions):

```
BoxWrongType.java:5: error: incompatible types: int cannot be converted to Double
BoxWrongType.java:6: error: incompatible types: int cannot be converted to Double
```

Autoboxing only goes to the **matching** wrapper: `int` to `Integer`, `double` to `Double`. The compiler will not widen and box in the same step, so 5 (an `int`) cannot become a `Double`. The fix is to supply a `double` value: write `Double price = 5.0;` and `printPrice("pencil", 2.0);`. Then the program compiles, and the call prints `pencil costs 2.0`. The same rule blocks the other direction: a `double` such as 7.5 cannot be boxed into an `Integer`.

PAUSE AND TRY

Question 3 Multiple choice

Which declaration compiles?

- A. `Double total = 10;`
- B. `Integer count = 3.0;`
- C. `Double average = 10.0;`
- D. `Integer score = 3.5;`

Question 4 Short answer

In `AutoboxParameter`, line 7 calls `printBadge("Grace", 9)`. The parameter `points` has type `Integer`, but `9` is an `int`. Explain what the compiler does so that this call works.

4.7.3 Unboxing

Autoboxing goes from a number to an object. The reverse is just as common. You have an `Integer` and you want to do math with it, store it in an `int`, or pass it to a method such as `Math.abs` that expects a plain `int`. Opening the envelope by hand would be tedious, so the compiler does it for you.

Unboxing is the automatic conversion from a wrapper object to its primitive value: an `Integer` to an `int`, and a `Double` to a `double`. It mirrors autoboxing, and the compiler applies it when a wrapper object is:

1. **assigned** to a variable of the matching primitive type, or
2. **passed as an argument** to a method whose parameter has the matching primitive type.

Java also unboxes a wrapper whenever you use it in arithmetic or a comparison such as `<`, because those operators work on primitive values.

KEY IDEA

Unboxing is the compiler automatically taking the `int` out of an `Integer`, or the `double` out of a `Double`, when the object is assigned to a primitive variable or passed to a primitive parameter.

WORKED EXAMPLE Unboxing by assignment

Each wrapper variable here is used where a primitive is expected.

```
1 Integer stored = 40;
2 int copy = stored;
3 int doubled = stored * 2;
4 Double rate = 1.5;
5 double rateCopy = rate;
6 System.out.println(copy);
7 System.out.println(doubled);
8 System.out.println(rateCopy);
```

Line	What happens	New variable
1	40 is autoboxed; stored refers to an <code>Integer</code> holding 40	
2	stored is assigned to an <code>int</code> , so it is unboxed to 40	copy is 40
3	<code>*</code> needs a primitive, so stored is unboxed; 40×2 is 80	doubled is 80
4	1.5 is autoboxed into a <code>Double</code>	
5	rate is assigned to a <code>double</code> , so it is unboxed to 1.5	rateCopy is 1.5

```
40
80
1.5
```

Now the second situation, method parameters, using `Math` methods from Topic 1.11:

```
1 Integer change = -8;
2 System.out.println(Math.abs(change));
3 Double area = 49.0;
4 System.out.println(Math.sqrt(area));
```

```
8
7.0
```

`Math.abs` has a version with an `int` parameter. `change` is an `Integer`, so on line 2 it is unboxed to `-8`, and `Math.abs` returns `8`. On line 4, `Math.sqrt` expects a `double`; `area` is unboxed to `49.0`, and the square root is `7.0`.

How do unboxing and the Topic 1.5 widening rule combine? Java can unbox and then widen, but it never narrows:

Code	Compiles?	Why
<code>Integer count = 7; int n = count;</code>	yes	Unboxing to the matching type, <code>int</code>
<code>Integer count = 7; double d = count;</code>	yes, <code>d</code> is <code>7.0</code>	Unbox to <code>int</code> <code>7</code> , then widen to <code>double</code> (safe)
<code>Double rating = 4.0;</code> <code>int stars = rating;</code>	no	Unbox to <code>double</code> , but a <code>double</code> never narrows to <code>int</code> without a cast

The second row prints `7.0`:

```
1 Integer count = 7;
2 double asDouble = count;
3 System.out.println(asDouble);
```

The third row gives the compile error `incompatible types: Double cannot be converted to int`, for the same reason `int stars = 4.0;` fails in Topic 1.5.

WATCH OUT

Unboxing `null`. An `Integer` variable can hold `null`, for instance when a score has not been recorded yet. Unboxing needs an object to take the number out of:

```
1 Integer bestScore = null;
2 System.out.println("Looking up best score");
3 int score = bestScore;
4 System.out.println(score);
```

Line 2 prints, and then line 3 crashes:

```
Looking up best score
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "<local1>" is null
```

(The exact message varies between Java versions.) Line 3 looks like an ordinary assignment, so students do not expect a crash there. But to unbox, Java calls a method on the `Integer` object to get its value, and calling a method through `null` throws a `NullPointerException`, just as in Topic 1.14. The compiler cannot catch this, because the type is correct; the problem is the value at run time. The fix is to make sure the variable refers to an object before it is unboxed, for example by checking first:

```
1 Integer bestScore = null;
2 System.out.println("Looking up best score");
3 if (bestScore != null)
4 {
5     int score = bestScore;
6     System.out.println(score);
7 }
8 else
9 {
10    System.out.println("No score yet");
11 }
```

```
Looking up best score
No score yet
```

PAUSE AND TRY

Question 5 Tracing

Write the exact output.

```
1 Integer start = -6;
2 Integer end = 4;
3 int distance = Math.abs(start) + end;
4 Double scale = 2.5;
5 double scaled = scale * distance;
6 System.out.println(distance);
7 System.out.println(scaled);
```

Question 6 Debugging

This program prints `Base: 50` and then crashes with a `NullPointerException` on line 8. Explain why and correct line 5 so that the program prints a total of 50.

BonusNull.java

```
1 public class BonusNull
2 {
3     public static void main(String[] args)
4     {
5         Integer bonus = null;
6         int base = 50;
7         System.out.println("Base: " + base);
8         int total = base + bonus;
9         System.out.println("Total: " + total);
10    }
11 }
```

4.7.4 Wrapper Objects Are Immutable

In Topic 3.6, a method that received a `Player` could call `addPoints` and the caller saw the change. Can the same thing happen with an `Integer`? Is there any way to change the number inside an `Integer` object?

There is not. An `Integer` object is **immutable**: once it is created, its value can never change. The same is true for every `Double` object. You saw this idea with `String` in Topic 1.15. Neither wrapper class has a single method that changes the stored number; the envelope is sealed for good.

But then how does `visits = visits + 1;` work when `visits` is an `Integer`? Break it into steps:

1. `visits` is unboxed to its `int` value.
2. 1 is added, giving a new `int`.
3. That `int` is autoboxed into a **brand-new** `Integer` object.
4. `visits` is changed to refer to the new object.

The old object is not changed. The variable just moves to a different object.

KEY IDEA

An `Integer` or `Double` object can never change. Code that seems to change a wrapper variable really builds a new object and makes the variable refer to it.

WORKED EXAMPLE An alias that does not follow along

Two `Integer` variables start out referring to the same object.

```
1 Integer visits = 3;
2 Integer saved = visits;
3 visits = visits + 1;
4 System.out.println(visits);
5 System.out.println(saved);
```

- Line 1: an `Integer` holding 3 is created (call it object A), and `visits` refers to it.
- Line 2: `saved` gets a copy of the reference, so `visits` and `saved` are aliases of object A (Topic 2.6).

- Line 3: `visits + 1` unboxes A's 3 and adds 1. The result, 4, is boxed into a new object B, and `visits` now refers to B.

After line	visits refers to	saved refers to
2	A (3)	A (3)
3	B (4)	A (3)

4
3

Compare this with a mutable `Player` from Topic 3.6: if two variables referred to the same `Player` and you called `addPoints` through one, both would see the change. With an `Integer` there is no mutator, so the only way to get 4 is a new object, and `saved` keeps the old one.

The same reasoning explains what happens when an `Integer` is passed to a method:

`IntegerParameter.java`

```

1 public class IntegerParameter
2 {
3     public static void main(String[] args)
4     {
5         Integer score = 10;
6         addBonus(score);
7         System.out.println("Back in main: " + score);
8     }
9
10    public static void addBonus(Integer points)
11    {
12        points = points + 5;
13        System.out.println("Inside addBonus: " + points);
14    }
15 }
```

Inside addBonus: 15
Back in main: 10

On line 6, `points` receives a copy of the reference to the `Integer` holding 10. Line 12 cannot change that object, so it builds a new `Integer` holding 15 and stores its reference in `points`. As Topic 3.6 showed, assigning to a parameter changes only the parameter. `score` in `main` still refers to the object holding 10.

WATCH OUT

Expecting a method to update the caller's `Integer`. Because an `Integer` is an object, students sometimes think it behaves like a `Player` and that `addBonus(score)` will change `score`. It cannot: the object is immutable, and the parameter is a separate variable. If a method needs to produce a new number, have it **return** the value and let the caller store it:

```
1 public static Integer addBonus(Integer points)
2 {
3     return points + 5;
4 }
```

With the call written as `score = addBonus(score);`, the program prints Back in main: 15.

Comparing wrapper values

Because wrapper variables hold references, take care with `==`. When one side of `==` is a primitive `int`, Java unboxes the `Integer` and compares the numbers. When **both** sides are `Integer` objects, `==` compares references, exactly like `==` on two `String` variables in Topic 2.6: it asks whether they are the same object, not whether they hold the same number. Whether two `Integer` variables holding equal numbers share one object depends on how Java decided to create the objects, which you cannot see in the code, so `==` between two wrapper objects is not a reliable test of their values. Compare the values instead, either with `equals` (as with strings in Topic 1.15) or by unboxing into `int` variables first:

```
1 Integer stored = 1000;
2 int goal = 1000;
3 System.out.println(stored == goal);
4 Integer first = 1000;
5 Integer second = 1000;
6 System.out.println(first.equals(second));
7 int firstValue = first;
8 int secondValue = second;
9 System.out.println(firstValue == secondValue);
```

```
true
true
true
```

Line 3 compares an `Integer` with an `int`, so `stored` is unboxed. Line 6 uses `equals` to compare two `Integer` objects by value. Lines 7 and 8 unbox both objects, so line 9 compares two `int` values.

PAUSE AND TRY

Question 7 Tracing

Write the exact output.

ImmutableTrace.java

```
1 public class ImmutableTrace
2 {
3     public static void main(String[] args)
4     {
5         Integer tickets = 8;
6         Integer backup = tickets;
7         tickets = tickets - 3;
8         sellOne(tickets);
9         System.out.println(tickets);
10        System.out.println(backup);
11    }
12
13    public static void sellOne(Integer count)
14    {
15        count = count - 1;
16        System.out.println(count);
17    }
18 }
```

Question 8 Multiple choice

In the alias example, `saved` still prints 3 after `visits = visits + 1`; . Which explanation is correct?

- A. Line 2 made `saved` a separate copy of the `Integer` object, so it was never connected to `visits`.
- B. `visits + 1` built a new `Integer` object for `visits`; the object `saved` refers to was never changed.
- C. Java changed the shared object to 4, but `saved` remembers the value it had when it was assigned.
- D. Wrapper variables hold primitive values, not references, so assigning one to another copies the number.

4.7.5 Turning Strings into Numbers

In Topic 4.6 you read lines from a text file and used `split` to break each line into pieces. Every piece is a `String`, even when it looks like a number. If a line says `Ada,12,88.5`, the piece `"12"` is two characters, not the number twelve. You cannot do math with it: as Topic 1.15 showed, `+` with strings joins them.

```
1 String first = "12";
2 String second = "30";
3 System.out.println(first + second);
4 int sum = Integer.parseInt(first) + Integer.parseInt(second);
5 System.out.println(sum);
```

```
1230
42
```

Line 3 joins "12" and "30" into "1230". Line 4 first converts each string to an `int`, so `+` adds: 12 + 30 is 42.

The conversion on line 4 is done by a static method of the `Integer` class:

```
1 public static int parseInt(String s)
```

`Integer.parseInt` takes a `String` that spells a whole number and returns that number as an `int`. The `Double` class has a partner method:

```
1 public static double parseDouble(String s)
```

`Double.parseDouble` takes a `String` that spells a number and returns it as a `double`. Both are class methods (Topic 3.7), so you call them through the class name, `Integer.parseInt(...)` and `Double.parseDouble(...)`, never through an object. Use them whenever a number arrives as text: a piece of a file line, a word from a `Scanner`'s `next()`, or any other `String`.

KEY IDEA

`Integer.parseInt` turns a `String` such as "12" into the `int` 12, and `Double.parseDouble` turns a `String` such as "88.5" into the `double` 88.5. Use them whenever numbers arrive as text.

WORKED EXAMPLE Taking apart a record

One line of a file holds a player's name, goals, and shooting percentage. Watch the type of each piece.

```
1 String line = "Ada,12,88.5";
2 String[] parts = line.split(",");
3 String name = parts[0];
4 int goals = Integer.parseInt(parts[1]);
5 double percent = Double.parseDouble(parts[2]);
6 System.out.println(name + " next season: " + (goals + 1));
7 System.out.println(percent / 2);
```

Line	Expression	Type	Value
2	<code>parts[0], parts[1], parts[2]</code>	<code>String</code>	"Ada", "12", "88.5"
4	<code>Integer.parseInt(parts[1])</code>	<code>int</code>	12
5	<code>Double.parseDouble(parts[2])</code>	<code>double</code>	88.5
6	<code>goals + 1</code>	<code>int</code>	13
7	<code>percent / 2</code>	<code>double</code>	44.25

The parentheses around `goals + 1` on line 6 make Java add the numbers before joining the result to the string.

```
Ada next season: 13
44.25
```

Now vary the text. `parseDouble` accepts a whole number, and `parseInt` accepts a minus sign:

```
1 double price = Double.parseDouble("90");
2 int count = Integer.parseInt("-5");
3 System.out.println(price);
4 System.out.println(count);
```

```
90.0
-5
```

`Double.parseDouble("90")` returns the `double` 90.0, which prints with a decimal point.
`Integer.parseInt("-5")` returns the negative `int` -5.

WATCH OUT

Asking `parseInt` for something that is not a whole number. Hours worked are often written with a decimal, and it is easy to reach for `parseInt` out of habit:

```
1 String hours = "7.5";
2 System.out.println("Reading hours");
3 int worked = Integer.parseInt(hours);
4 System.out.println(worked);
```

The program prints `Reading hours`, then crashes on line 3:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "7.5"
```

(The wording can vary between Java versions.) `parseInt` does not round or truncate the way a cast does in Topic 1.5. If the string is not exactly a whole number, it throws a `NumberFormatException` at run time. The compiler cannot catch this, because any `String` is a legal argument; the problem is the text inside it. The same crash happens for text with a space in it, such as `" 14"`, or a word such as `"twelve"`. The fix is to use the method that matches the text: `double worked = Double.parseDouble(hours);` prints 7.5.

PAUSE AND TRY

Question 9 Tracing

Write the exact output.

```
1 String[] cells = {"8", "15", "2.5"};
2 System.out.println(cells[0] + cells[1]);
3 int total = Integer.parseInt(cells[0]) + Integer.parseInt(cells[1]);
4 System.out.println(total);
5 double result = total * Double.parseDouble(cells[2]);
6 System.out.println(result);
```

Question 10 Debugging

This program prints `Player: Lee` and then crashes on line 8 with

`NumberFormatException: For input string: " 14"`. Explain the error and correct line 6.

SplitSpace.java

```
1 public class SplitSpace
2 {
3     public static void main(String[] args)
4     {
5         String line = "Lee, 14";
6         String[] parts = line.split(",");
7         System.out.println("Player: " + parts[0]);
8         int points = Integer.parseInt(parts[1]);
9         System.out.println("Points: " + points);
10    }
11 }
```

Putting it together

A weather station saves one line per day in `weather.txt`: the day, the overnight low in whole degrees, and the rainfall in inches.

```
Mon,-3,0.5
Tue,4,1.25
Wed,-7,0.0
Thu,2,2.75
```

This program finds the warmest night, adds up the rain, and prints each low through a method with an `Integer` parameter.

WeatherLog.java

```
1 import java.io.File;
2 import java.io.IOException;
3 import java.util.Scanner;
4
5 public class WeatherLog
6 {
7     public static void main(String[] args) throws IOException
8     {
9         Scanner file = new Scanner(new File("weather.txt"));
10        int warmestLow = Integer.MIN_VALUE;
11        String warmestDay = "";
12        double totalRain = 0.0;
13        while (file.hasNext())
14        {
15            String[] parts = file.nextLine().split(",");
16            int low = Integer.parseInt(parts[1]);
17            double rain = Double.parseDouble(parts[2]);
18            printLow(parts[0], low);
19            if (low > warmestLow)
20            {
21                warmestLow = low;
22                warmestDay = parts[0];
23            }
24        }
25    }
26 }
```

```

24         totalRain += rain;
25     }
26     file.close();
27     System.out.println("Warmest night: " + warmestDay);
28     System.out.println("Total rain: " + totalRain);
29 }
30
31 public static void printLow(String day, Integer low)
32 {
33     if (low < 0)
34     {
35         System.out.println(day + ": " + Math.abs(low) + " below zero");
36     }
37     else
38     {
39         System.out.println(day + ": " + low + " above zero");
40     }
41 }
42 }

```

Here is where each idea from the chapter appears:

Line(s)	Idea
10	<code>Integer.MIN_VALUE</code> is the starting maximum, so even a night below zero can be the warmest (4.7.1)
15–17	Each piece of the line is a <code>String</code> ; <code>parseInt</code> and <code>parseDouble</code> turn them into numbers (4.7.5)
18, 31	The <code>int low</code> is passed to an <code>Integer</code> parameter, so it is autoboxed (4.7.2)
33, 35	<code>low < 0</code> unboxes <code>low</code> for the comparison; <code>Math.abs(low)</code> unboxes it for an <code>int</code> parameter (4.7.3)
39	Joining the <code>Integer low</code> to a string shows its number

Tracing the loop:

Line read	low	rain	Printed by printLow	warmestLow	warmestDay	totalRain
(start)				-2147483648	""	0.0
Mon, -3, 0.5	-3	0.5	Mon: 3 below zero	-3	Mon	0.5
Tue, 4, 1.25	4	1.25	Tue: 4 above zero	4	Tue	1.75
Wed, -7, 0.0	-7	0.0	Wed: 7 below zero	4	Tue	1.75
Thu, 2, 2.75	2	2.75	Thu: 2 above zero	4	Tue	4.5

```

Mon: 3 below zero
Tue: 4 above zero
Wed: 7 below zero
Thu: 2 above zero
Warmest night: Tue
Total rain: 4.5

```

Without `parseInt` and `parseDouble`, the program would have only strings, which cannot be compared with `>` or added as numbers. Without `Integer.MIN_VALUE`, a starting value of 0 would

have worked for this file only by luck (Tuesday's low happens to be positive); a week where every night was below zero would report 0.

Common mistakes

Every mistake in this chapter shows up as a message. Use this table to look one up.

Message	Usual cause	Fix
int cannot be converted to Double (compile)	Boxing an int into a Double, such as <code>Double price = 5;</code>	Use a double value (5.0), or an Integer variable
double cannot be converted to Integer (compile)	Boxing a double into an Integer	Use a Double variable, or an int value
Double cannot be converted to int (compile)	Unboxing a Double into an int, which would narrow	Store it in a double
NullPointerException (run time)	Unboxing a wrapper variable that holds null	Make sure it refers to an object first
NumberFormatException (run time)	<code>parseInt</code> or <code>parseDouble</code> given text that is not a number (decimal point for <code>parseInt</code> , spaces, words)	Use the matching method, and split so pieces have no extra spaces

Vocabulary

Term	Definition	Example
wrapper class	A class whose objects each hold one primitive value, so the value can be used where an object is needed	Integer for int, Double for double
java.lang package	The group of classes available to every program without an import	Integer, Double, String, Math
autoboxing	The compiler's automatic conversion of an int to an Integer or a double to a Double	<code>Integer count = 12;</code>
unboxing	The compiler's automatic conversion of an Integer to an int or a Double to a double	<code>int n = count;</code>
immutable (review)	Describes an object whose value can never change after it is created	An Integer holding 3 always holds 3
Integer.parseInt	A class method that returns the int spelled by a String	<code>Integer.parseInt("12")</code> is 12
Double.parseDouble	A class method that returns the double spelled by a String	<code>Double.parseDouble("90")</code> is 90.0

NumberFormatException	The exception <code>parseInt</code> or <code>parseDouble</code> throws when the text is not a number of the right kind	<code>Integer.parseInt("7.5")</code>
------------------------------	--	--------------------------------------

Chapter summary

- `Integer` and `Double` are wrapper classes in `java.lang`: each object holds one `int` or `double`, so a number can go where only objects are allowed. `Integer.MIN_VALUE` and `Integer.MAX_VALUE` are class constants that make safe starting values for maximum and minimum algorithms.
- Autoboxing turns an `int` into an `Integer` or a `double` into a `Double` when the value is assigned to a wrapper variable or passed to a wrapper parameter. It only boxes into the matching wrapper.
- Unboxing takes the primitive value out of a wrapper when the object is assigned to a primitive variable, passed to a primitive parameter, or used in arithmetic or a comparison. Unboxing `null` throws a `NullPointerException`.
- Wrapper objects are immutable. Arithmetic on a wrapper variable builds a new object, so aliases and the caller's variables never see the change.
- `Integer.parseInt` and `Double.parseDouble` convert text into numbers; text that is not a number of the right kind throws a `NumberFormatException`.

Review questions

Question 11 Multiple choice

Which statement is true?

- A. A method with an `Integer` parameter can change the number inside the caller's `Integer` object.
- B. A program can call `Double.parseDouble` without any `import` statement.
- C. `Integer.parseInt` must be called on an `Integer` object, such as `count.parseInt("5")`.
- D. Autoboxing can turn an `int` value into a `Double` object.

Question 12 Multiple choice

The program prints `81` and `40.5`. Which description of the conversions on lines 6, 7, and 8 is correct?

ConversionSpot.java

```
1 public class ConversionSpot
2 {
3     public static void main(String[] args)
4     {
5         int raw = 9;
6         Integer boxed = raw;
7         int squared = square(boxed);
8         Double half = squared / 2.0;
9         System.out.println(squared);
10        System.out.println(half);
11    }
12
13    public static int square(int value)
14    {
15        return value * value;
16    }
17 }
```

- A. Line 6 autoboxes; line 7 unboxes; line 8 autoboxes.
- B. Line 6 unboxes; line 7 autoboxes; line 8 unboxes.
- C. Line 6 autoboxes; line 7 autoboxes; line 8 does neither.
- D. Only line 6 converts; lines 7 and 8 do neither.

Question 13 Tracing

The values arrive as strings. Write the exact output.

```
1 String[] entries = {"-4", "-9", "-1", "-6"};
2 System.out.println(entries[0] + entries[1]);
3 int highest = Integer.MIN_VALUE;
4 for (String entry : entries)
5 {
6     int value = Integer.parseInt(entry);
7     if (value > highest)
8     {
9         highest = value;
10    }
11 }
12 System.out.println(highest);
```

Question 14 Debugging

This program does not compile. Java reports

incompatible types: double cannot be converted to Integer for line 7. Explain the error and correct line 7.

HoursBug.java

```
1 public class HoursBug
2 {
3     public static void main(String[] args)
4     {
5         String line = "Lee,7.5";
6         String[] parts = line.split(",");
7         Integer hours = Double.parseDouble(parts[1]);
8         System.out.println(parts[0] + " worked " + hours);
9     }
10 }
```

Question 15 Short answer

In Topic 3.6, a method that received a `Player` could call `addPoints` and the caller's player gained points. A method that receives an `Integer` can never make the caller's `Integer` show a different number. Both parameters hold a copy of a reference. Explain why the results differ.

Question 16 Multiple choice

What does this statement print?

```
1 System.out.println(Double.parseDouble("7") + Integer.parseInt("3"));
```

- A. 10.0
- B. 10
- C. 73
- D. 7.03

Question 17 Tracing

Write the exact output.

```
1 Integer total = 7;
2 Integer count = 2;
3 double average = total / count;
4 System.out.println(average);
5 double better = 1.0 * total / count;
6 System.out.println(better);
```

Reflection

- ☐ I can explain how an `Integer` or `Double` object differs from an `int` or `double`, and why no import is needed. Reread: 4.7.1 The Integer and Double Classes
- ☐ I can choose `Integer.MIN_VALUE` or `Integer.MAX_VALUE` as the starting value for a maximum or minimum. Reread: 4.7.1 The Integer and Double Classes
- ☐ I can spot where Java autoboxes a primitive, and explain why an `int` cannot become a `Double`. Reread: 4.7.2 Autoboxing
- ☐ I can spot where Java unboxes a wrapper object, and explain why unboxing null crashes. Reread: 4.7.3 Unboxing

- ☐ I can explain why a method or an alias never sees a change to an Integer or Double. Reread: 4.7.4 Wrapper Objects Are Immutable
- ☐ I can convert text to numbers with Integer.parseInt and Double.parseDouble, and predict a NumberFormatException. Reread: 4.7.5 Turning Strings into Numbers

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

```
-3
-1
```

Visit	jump	jump > biggest?	biggest afterward	Printed
start			-2147483648	
1	-3	yes	-3	-3
2	-8	no	-3	
3	-1	yes	-1	-1
4	-6	no	-1	

The `println` is inside the `if`, so it prints only when `biggest` changes. The first element always replaces `Integer.MIN_VALUE`.

Reread 4.7.1 *The Integer and Double Classes* if your output differed.

Question 2

Model answer: `Integer` is in the `java.lang` package, which Java makes available to every program automatically, the same as `String` and `Math`. `File` is in the `java.io` package, which is not automatic, so it must be imported.

A complete answer names `java.lang` as the package of `Integer` and says that this package needs no import.

Reread 4.7.1 *The Integer and Double Classes* if your answer differed.

Question 3

C. `10.0` is a `double`, and a `double` autoboxes into its matching wrapper, `Double`.

- **A** is wrong: `10` is an `int`, and an `int` boxes only into an `Integer`. Java reports `int cannot be converted to Double`. This comes from expecting the Topic 1.5 widening rule to apply to wrappers.
- **B** is wrong: `3.0` is a `double` even though its value is whole, and a `double` cannot be boxed into an `Integer`.
- **D** is wrong for the same reason as B: Java reports `double cannot be converted to Integer`.

BoxOptions.java

```

1 public class BoxOptions
2 {
3     public static void main(String[] args)
4     {
5         Double total = 10;
6         Integer count = 3.0;
7         Double average = 10.0;
8         Integer score = 3.5;
9     }
10 }

```

Compiling all four together reports errors on lines 5, 6, and 8 only.

Reread 4.7.2 *Autoboxing* if you chose differently.

Question 4

Model answer: Because the parameter's type is `Integer`, the wrapper class that matches `int`, the compiler autoboxes the argument: it adds code that creates an `Integer` object holding 9. The parameter points then refers to that object, and line 13 prints `Grace has 9 points`.

A complete answer says that the conversion is automatic (no `new` in the code), that it happens because the value is passed to a parameter of the matching wrapper type, and that the parameter refers to a new `Integer` object.

Reread 4.7.2 *Autoboxing* if your answer differed.

Question 5

```

10
25.0

```

Line	What happens	Value
3	start unboxes to -6 for <code>Math.abs</code> , which returns 6; end unboxes to 4 for <code>+</code>	distance is 10
5	scale unboxes to 2.5; 2.5×10 is a double	scaled is 25.0

scaled is a double, so it prints with a decimal point: 25.0.

Reread 4.7.3 *Unboxing* if your output differed.

Question 6

`bonus` is an `Integer` that holds `null`. On line 8, `base + bonus` needs a primitive value, so Java tries to unbox `bonus`. There is no object to take a number from, so Java throws a `NullPointerException`. The compiler allowed it because `Integer` is a correct type for `+`; only the value is wrong.

Corrected line 5:

```

1 Integer bonus = 0;

```

Now `bonus` refers to an `Integer` holding 0, line 8 unboxes it, and the program prints `Base: 50` and `Total: 50`. (Declaring `bonus` as an `int` would also work, because an `int` can never be `null`.)

Reread 4.7.3 *Unboxing* if your fix differed.

Question 7

4
5
8

Line	What happens	tickets	backup	count
5	Integer holding 8 created	8		
6	backup becomes an alias of the same object	8	8	
7	8 - 3 is boxed into a new object; only tickets moves to it	5	8	
8	count gets a copy of tickets's reference	5	8	5
15	5 - 1 is boxed into a new object; only count moves	5	8	4
16	prints 4	5	8	4
9-10	prints 5, then 8	5	8	

Neither line 7 nor line 15 changes an existing object, so `backup` keeps 8 and the method cannot change `tickets`.

Reread 4.7.4 *Wrapper Objects Are Immutable* if your output differed.

Question 8

B. `Integer` objects are immutable. Line 3 unboxes, adds, and boxes the result into a new object, then moves only `visits` to it. `saved` still refers to the original object holding 3.

- **A** is wrong: assigning one reference variable to another copies the reference, not the object. Right after line 2, both variables refer to the same object.
- **C** is wrong: an `Integer` object can never be changed, and a reference variable does not "remember" old values; it shows whatever its object holds.
- **D** is wrong: `Integer` is a class type, so its variables hold references. That is why they can also hold `null`.

Reread 4.7.4 *Wrapper Objects Are Immutable* if you chose differently.

Question 9

815
23
57.5

- Line 2 joins two strings: `"8" + "15"` is `"815"`.
- Line 3 converts both to `int` first: `8 + 15` is 23.
- Line 5: `Double.parseDouble("2.5")` is 2.5, and `23 × 2.5` is 57.5.

Reread 4.7.5 *Turning Strings into Numbers* if your output differed.

Question 10

The line has a space after the comma. Splitting on `","` alone gives `parts[0]` as `"Lee"` and `parts[1]` as `" 14"`, with the space still attached. `parseInt` accepts only the characters of a whole number, so the space makes it throw a `NumberFormatException`.

Corrected line 6:

```
1 String[] parts = line.split(", ");
```

Splitting on a comma followed by a space leaves `parts[1]` as `"14"`, and the program prints `Player: Lee and Points: 14.`

Reread [4.7.5 Turning Strings into Numbers](#) if your fix differed.

Question 11

B. `Double` is in the `java.lang` package, which every program can use without importing it.

- **A** is wrong: `Integer` objects are immutable. A method can only make its own parameter refer to a new object, which the caller never sees.
- **C** is wrong: `parseInt` is a static (class) method, so it is called through the class name, `Integer.parseInt("5")`.
- **D** is wrong: autoboxing goes only to the matching wrapper, so an `int` becomes an `Integer`, never a `Double`.

Reread [4.7.1 The Integer and Double Classes](#), [4.7.2 Autoboxing](#), [4.7.4 Wrapper Objects Are Immutable](#), and [4.7.5 Turning Strings into Numbers](#) for the choices you were unsure of.

Question 12

A.

- Line 6 assigns the `int` `raw` to an `Integer` variable: autoboxing.
- Line 7 passes the `Integer` `boxed` to `square`, whose parameter on line 13 is an `int`: unboxing. The method returns the `int` `81`.
- Line 8 computes `81 / 2.0`, a `double` `40.5`, and assigns it to a `Double` variable: autoboxing.

The other choices:

- **B** is wrong: it reverses every direction. Boxing goes from primitive to object; unboxing goes from object to primitive.
- **C** is wrong: line 7 sends an object to a primitive parameter, which is unboxing, and line 8 does box a `double` into a `Double`.
- **D** is wrong: a conversion happens every time a value crosses between a primitive type and its wrapper, including at a method parameter.

Reread [4.7.2 Autoboxing](#) and [4.7.3 Unboxing](#) if you chose differently.

Question 13

```
-4-9
-1
```

Line 2 joins two strings: `"-4"` followed by `"-9"` is `"-4-9"`. No subtraction happens, because the operands are strings.

entry	value	value > highest?	highest afterward
(start)			-2147483648
"-4"	-4	yes	-4
"-9"	-9	no	-4
"-1"	-1	yes	-1

"-6"	-6	no	-1
------	----	----	----

Every value is negative, so starting at `Integer.MIN_VALUE` matters; a start of 0 would print 0.

Reread *4.7.5 Turning Strings into Numbers* and *4.7.1 The Integer and Double Classes* if your output differed.

Question 14

`Double.parseDouble` returns a `double` (7.5). Assigning it to an `Integer` variable would need autoboxing, but a `double` can be boxed only into its matching wrapper, `Double`.

Corrected line 7:

```
1 Double hours = Double.parseDouble(parts[1]);
```

Now the `double` 7.5 is autoboxed into a `Double`, and the program prints `Lee worked 7.5`. (Declaring `hours` as a `double` would also work.) Do not "fix" it by switching to `Integer.parseInt`: that compiles, but "7.5" is not a whole number, so it would throw a `NumberFormatException` when the program runs.

Reread *4.7.2 Autoboxing* and *4.7.5 Turning Strings into Numbers* if your fix differed.

Question 15

Model answer: In both cases the parameter refers to the caller's object. A `Player` is mutable, so the method can call a mutator that changes that shared object, and the caller sees the change. An `Integer` is immutable: it has no method that changes its value. The only thing the method can do is compute a new number, which is boxed into a new `Integer` object and assigned to the parameter. Assigning to a parameter changes only the parameter, so the caller's variable still refers to the original, unchanged object.

A complete answer includes:

- Both parameters hold a copy of the caller's reference.
- `Player` can be changed through a mutator; `Integer` is immutable.
- Arithmetic on the parameter creates a new object and changes only the parameter.

Reread *4.7.4 Wrapper Objects Are Immutable* if your answer differed.

Question 16

A. `Double.parseDouble("7")` returns the `double` 7.0 and `Integer.parseInt("3")` returns the `int` 3. Both operands are numbers, so `+` adds. A `double` plus an `int` is a `double` (Topic 1.5), so the result is 10.0.

- **B** is wrong: it forgets that one operand is a `double`, which makes the sum a `double` that prints with a decimal point.
- **C** is wrong: it treats the values as strings and joins them. After parsing, there are no strings left in the expression.
- **D** is wrong: it joins "7.0" and "3" as text, which also assumes `+` is working on strings.

Reread *4.7.5 Turning Strings into Numbers* if you chose differently.

Question 17

3.0
3.5

- Line 3: `total / count` unboxes both to `int` values, 7 and 2. Dividing two `int` values truncates (Topic 1.3), so the result is the `int` 3. Only then is it widened to the `double` 3.0 for `average`. Wrapping the numbers in objects did not change how `int` division works.
- Line 5: `1.0 * total` unboxes `total` and multiplies it by a `double`, giving 7.0. Then `7.0 / 2` is a `double` division, 3.5.

Reread 4.7.3 *Unboxing* if your output differed.