






## Composing More Than Two Graphics Language


The `+` **operator** has two **operands**. It can add two numbers. What about adding **more than two** numbers? Write down the expression you use to add the **three** numbers 10, 20, and 30:

If you want to add 10 numbers, how many `+` operators do you need?

The **methods** `beside`, `above`, and `overlay` have two **parameters**. They can compose two graphics. What about composing **more than two** graphics? Write down the expression you use to produce the following graphics (assume the top ellipse is 80-by-40, and the bottom ellipse 40-by-80):

	
---	--

Can you write a different expression that produces an equal graphic?

	
---	--

How do we call a **binary operation**  $\oplus$  (a **method** with two **parameters**, an **operator** with two **operands**) where  $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ ?

- associative**     
  **commutative**     
  **distributive**

In the following table, mark cells with an **X** to indicate operators or methods that are commutative or associative:

	<code>n + n</code>	<code>n - n</code>	<code>above(g, g)</code>	<code>beside(g, g)</code>	<code>overlay(g, g)</code>
commutative					
associative					

It is quite common that we want to compose more than two graphics. Write an `above3` method that can place any given three graphics above each other (add this method to your `Toolbelt` class):

```

public static Graphic above3(Graphic top, Graphic mid, Graphic bot) {

```

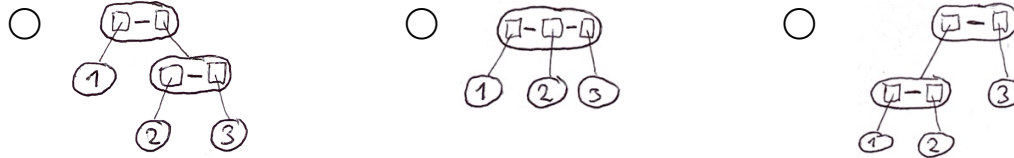


### Evaluating an Expression Language

What is the value of  $1 - 2 - 3$ ?

- $1 - 2 - 3 = (1 - 2) - 3 = -1 - 3 = -4$
- $1 - 2 - 3 = 1 - (2 - 3) = 1 - -1 = 2$

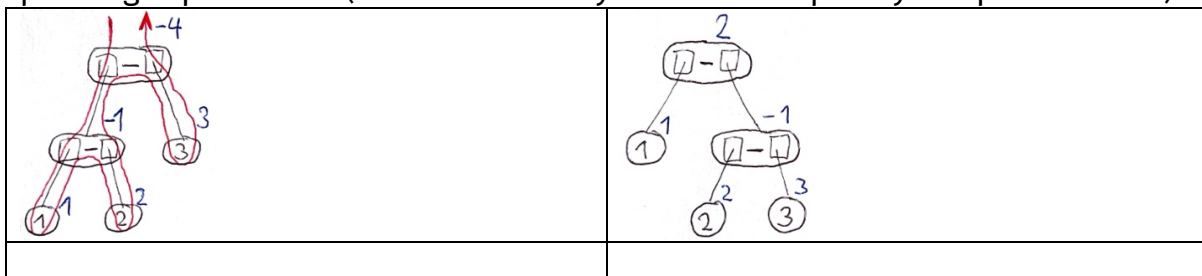
Which of the following trees correctly represents the expression  $1 - 2 - 3$ ?



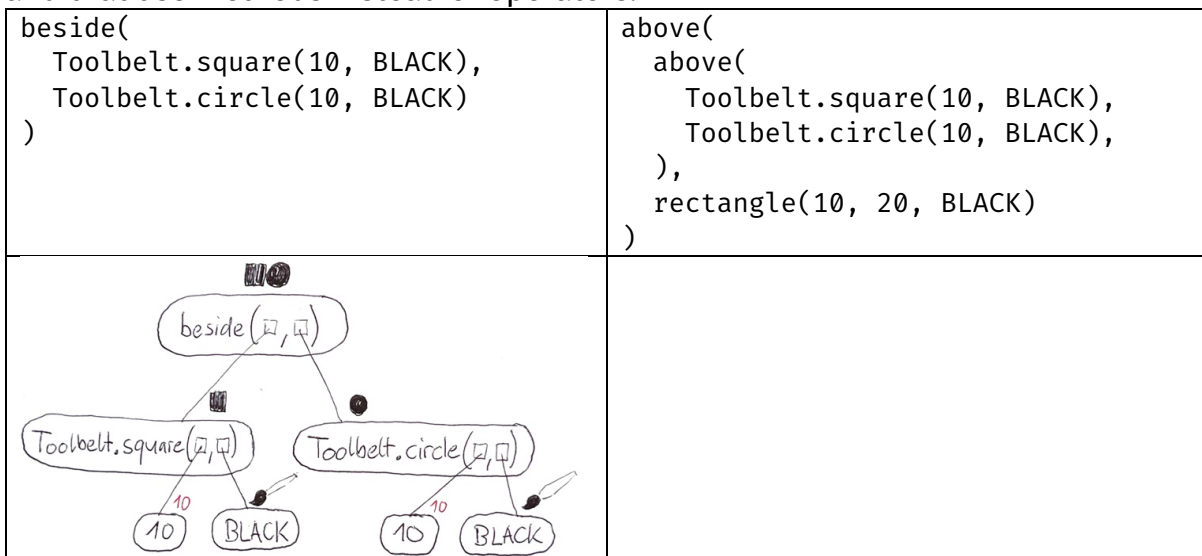
In this course, if we represent an expression as a tree, we want to make sure that **every** subexpression (every piece of the expression that can itself produce a value, e.g., an intermediate result) is represented as a subtree. The middle tree above does **not** fulfill this. We need to break its root node into multiple nodes (the incorrect root node in the middle tree combines the application of **two** - operators; each application produces a value and should be its own node).

We can use the expression tree to **evaluate** the expression (to determine the **value** the expression produces). For this, we go over the tree with a **depth-first post-order traversal**, as shown with the orange line in the left example below, such that we get the value for a subtree when we leave the root of that subtree.

Here are two trees, annotated with the value of each subtree. What are the corresponding expressions? (For one of them you need to explicitly use parentheses.)



Let's do the same for some expressions that produce graphics instead of numbers, and that use methods instead of operators:





## Types Language

We've casually used the word "type" many times so far. More precisely, we can see a **type** as a set of **values**. A type specifies all the values that it represents.

In Java, the built-in type with the smallest number of elements (smallest number of values) is **boolean**. It contains only two values: `true` and `false`.

The type **int** represents integer numbers. Watch out! Unlike in math, where the set of all integer numbers,  $\mathbb{Z}$ , is infinite, the Java type `int` only offers a finite number of values: `-2147483648 ... 2147483647`. Integer numbers that are smaller or larger than that cannot be represented as an `int`. In Java, we often use the type `int` when we want to count something, as long as the count doesn't get too large.

Can one represent the number of humans on planet Earth in an `int`?

yes

no

Can one represent the number of bytes of RAM of your computer in an `int`?

yes

no

The type **double** represents real numbers. Watch out! Unlike in math, where the set of all real numbers,  $\mathbb{R}$ , is infinite, the Java type `double` only offers a finite number of values. Values of type `double` are stored as "double-precision floating-point" numbers, using 64 bits. Many numbers cannot be represented exactly in a `double`, and thus calculating with `double` values leads to rounding errors.

Draw an expression tree of the Java expression `0.1 + 0.2 == 0.3`, and label each node with its value:

Compare the values you wrote above with the values you get on your computer.

Here are some of the types we encountered so far, plus a new one, `char`, for representing individual characters. For each **type**, provide two **expressions** that evaluate to **values** of that type: (You will complete the "Primitive?" column next week.)

Type:	Primitive?	Expression:	Expression:
<code>boolean</code>		<code>true</code>	<code>false &amp; !true</code>
<code>int</code>			
<code>double</code>			
<code>char</code>		<code>'A'</code>	
<code>String</code>			
<code>Color</code>			
<code>Point</code>			
<code>Graphic</code>			



## Literals: Values Embedded in Code Language

For some types in Java, one can write a value of that type literally in code.

Type	Some Literals
boolean	true false
int	0 1 1000000 2147483647
char	'A' 'Z' '0' '7' 'a' '-'
double	0.0 100.0 0.00001 1234567890.1234567890
String	"Hello" "Hi there!" "---"

**Literals** are **expressions** whose evaluation is trivial because their value is already directly written in code.

An interesting detail is that Java only has positive numeric literals. If you want to represent a negative number, you **have to** use the unary minus (i.e., -) operator in front of a positive literal. Draw the expression tree (with values and types) for:

-123

Which rows contain a legal Java literal, and what is its type?

Literal:	Legal Literal?	Type:
True		
2147483647		
100000000000000000		
3.1415		
100000000000000000.0		
-1		
'A'		
'ABC'		
' '		
' '		
' '		
' '		
' '		
' '		
"A"		
" "		
" "		
" "		
" "		
" "		

## Parentheses Language

We have seen **parentheses** that are used to “group” expressions, e.g., in  $1-(2-3)$ . Which claims are true?

- We can **always** place parentheses around an expression, without changing its meaning.
- In a piece of Java code, every pair of parentheses “groups” an expression.



## Conditional Computation: Ternary Operator `c ? t : e` Language

Sometimes we have to take decisions. Under some conditions we need to produce some value, otherwise we need to produce some other value. In Java we can use the **conditional operator** for this. This is also called the **ternary operator**, because it has **three** operands (it is the only three-operand operator in Java).

The following expression produces the color red if `value` is negative, and the color black otherwise:

```
value < 0 ? RED : BLACK
```

Let's wrap this expression in a method:

```
public static Color accountingColor(double value) {  
    return value < 0 ? RED : BLACK;  
}
```

Write a method that produces an upward-pointing triangle if the value is positive or zero, and a downward-pointing (i.e., rotated) triangle if the value is negative.

```
public static Graphic upOrDownIndicator(int value) {
```

Write a method that produces the String "even" if the value is even, and the String "odd" if the value is odd (hint: in Java, the modulo operator `a % d` produces the remainder of the division `a / d`; that is, `a % 2` will produce either 0 or 1):

```
public static String evenOrOdd(int value) {
```

Each of the three operands of the `?:` operator is an expression. The first operand is a **condition**, an expression that evaluates to `true` or to `false`. The value of this operand decides the result of the conditional. If the condition is `true`, the conditional produces the result of the second operand. If the condition is `false`, the conditional produces the result of the third operand.

The **type** of the condition must be `boolean`. The second and third operand can have any type (e.g., `Color`, `Graphic`, `String`, `double`, or `int`), but these two operands must have the **same** type. The type of the entire conditional expression corresponds to the type of the second and third operands – after all, the expression produces what the second or third operand produces.

Write a method that produces "Times" if a serif font is requested, and "Arial" otherwise:

```
public static String fontName(boolean serif) {  
    return serif ?  
}
```



## Type-Checking an Expression Language

When compiling code, a compiler first parses the code (builds a tree), and then, for every expression, it checks that it's plugged together correctly in terms of types. This is called **type-checking**. If type-checking fails, you receive a **type error** when compiling the program.

Which of the following expressions is type-correct? Complete the table:

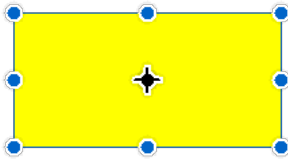
Expression (Source) & Expression Tree	Type Correct	Explain
1 + 2 	Yes	1 is a literal of type int. 2 is a literal of type int. There is a + operator expecting two ints, and it produces a result of type int.
1 + true 	No	1 is a literal of type int. true is a literal of type boolean. There is no + operator expecting an int and a boolean.
"Hello" / 2 		
1 / 2 * 3 	Yes	1 is a literal of type int. 2 is a literal of type int. There is a / operator expecting two ints, and it produces a result of type int. 3 is a literal of type int. There is a * operator expecting two ints, and it produces a result of type int.
1 < 2 < 3		
rgb(255, 128, 0)		255 is a literal of type int. 128 is a literal of type int. 0 is a literal of type int. There is an rgb method expecting three ints, producing a result of type Color
rectangle(1.0, 2.0, true ? RED : GREEN)		1.0 is a literal of type double.



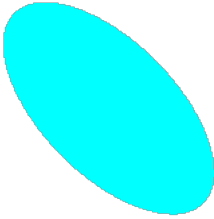
## Bounding Box Library

Each graphic has a **tight, axis-aligned bounding box**: the rectangle with sides parallel to the horizontal and vertical axes enclosing the graphic as tightly as possible.

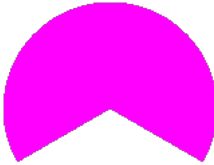
Draw the bounding boxes around the following graphics. Then indicate the nine points defined by the corners and midpoints of the bounding box:



```
rectangle(200, 100, YELLOW)
```



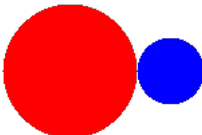
```
rotate(45, ellipse(200, 100, CYAN))
```



```
rotate(
  -30,
  circularSector(80, 240, MAGENTA)
)
```



```
rotate(
  30,
  text("upwards", "Helvetica", 40, GREEN)
)
```



```
beside(
  Toolbelt.circle(100, RED),
  Toolbelt.circle(50, BLUE)
)
```



```
above(
  circularSector(50, 210, MAGENTA),
  circularSector(50, 210, CYAN)
)
```



```
Toolbelt.beside3(
  rotate(0, Toolbelt.square(50, BLACK)),
  rotate(15, Toolbelt.square(50, BLACK)),
  rotate(45, Toolbelt.square(50, BLACK))
)
```



The last two examples show composite graphics where the component graphics don't touch each other. E.g., the `above` method composes the two circular sectors into a graphic such that their two bounding boxes touch. Two bounding boxes touching each other does not mean that the two graphics inside those two bounding boxes will also touch each other.





## Compose and the Invisible Pinning Position Library

Besides `beside`, `above`, and `overlay`, JTamaro also offers a fourth binary composition method:

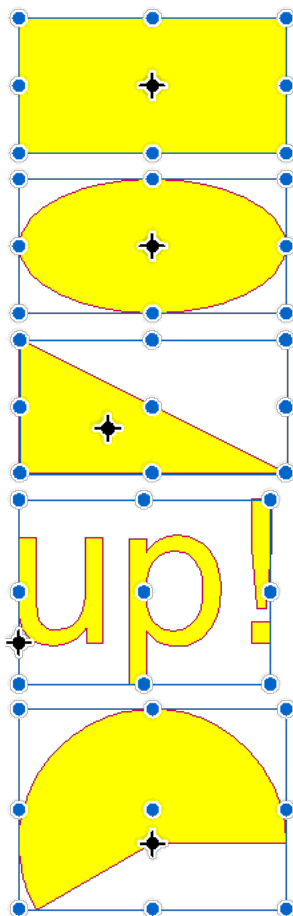
```
Graphic compose(Graphic top, Graphic bottom)
```

Let's see what `compose` produces, and how it compares to `overlay`:

<code>compose(wide(), tall())</code>	<code>overlay(wide(), tall())</code>

They seem to do the same thing! But there **is** a difference.

To see it, we need to understand a previously invisible aspect of graphics: each graphic has a **pinning position**. Here are the primitive graphics with their bounding box (blue axis-aligned rectangle enclosing the graphic), the nine points defined by the bounding box (corners, midpoints), and their pinning positions (black):



```
rectangle(200, 100, YELLOW)
```

```
ellipse(200, 100, YELLOW)
```

```
triangle(200, 100, 90, YELLOW)
```

```
text("up!", "Helvetica", 150, YELLOW)
```

```
circularSector(100, 210, YELLOW)
```

Sometimes the pinning position lies on a bounding box point, sometimes not.



## Creating a Graphic with a Different Pinning Position Library

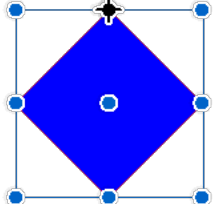
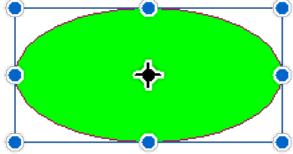
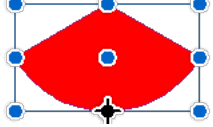
The `compose` method overlays two graphics at their pinning positions. If you want to control how they get aligned, you can first create graphics with the necessary pinning positions and then compose those.

```
Graphic pin(Point point, Graphic graphic)
```

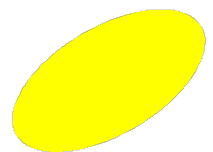
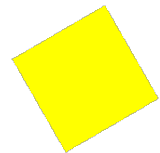
JTamaro provides some **names** for the nine points defined by a bounding box. These names denote **values** of **type** `Point`:

TOP_LEFT	TOP_CENTER	TOP_RIGHT
CENTER_LEFT	CENTER	CENTER_RIGHT
BOTTOM_LEFT	BOTTOM_CENTER	BOTTOM_RIGHT

Write the expressions that generate the described graphics. Consider the pinning position of a primitive graphic. There might be no need to call `pin` at all.

	<pre>pin(TOP_CENTER,     rotate(45,         Toolbelt.square(100, BLUE)     ) )</pre>
 <p>A green 200-by-100 ellipse pinned in its center.</p>	
 <p>A red circular sector with radius 80 and a 120-degree angle that is rotated so its tip faces upward, and pin it at the bottom center.</p>	

For the following expressions, draw the pinning position on the graphic:

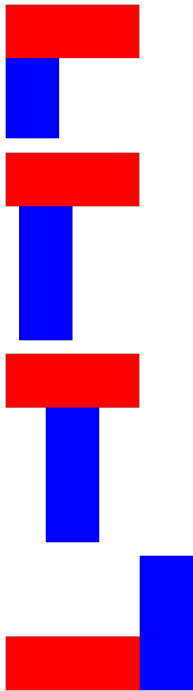
	<pre>pin(BOTTOM_CENTER,     rotate(30,         ellipse(200, 100, YELLOW)     ) )</pre>
	<pre>rotate(180,     pin(BOTTOM_CENTER,         rotate(30,             Toolbelt.square(100, YELLOW)         )     ) )</pre>

You can imagine that for doing `rotate`, JTamaro pushes a pin at the pinning position and rotates the graphic around that pin.



### Pin and Compose – General and Powerful Library

Using wide, tall, pin, and compose, write expressions that create these graphics:



	<pre>compose(   pin(TOP_LEFT, wide()),   pin(TOP_LEFT, tall()) )</pre>
	<pre>compose(</pre>
	<pre>compose(</pre>
	<pre>compose(</pre>

Draw the graphic produced by the following expression:

	<pre>compose(   pin(BOTTOM_RIGHT, wide()),   pin(TOP_LEFT, tall()) )</pre>
	<pre>compose(   pin(TOP_LEFT, tall()),   pin(BOTTOM_RIGHT, wide()) )</pre>

Draw the **bounding boxes** and the **pinning position** of each of the above composite graphics (i.e., all six results of calling compose on this page).

compose is a more **general** composition operation than beside, above, or overlay. Why? Because if all we had was compose and pin, we could implement beside, above, and overlay ourselves! The graphics produced by beside, above, and overlay have their pinning positions in their center. Here is how we could implement our own beside method:

```
public static Graphic myBeside(Graphic left, Graphic right) {
  return pin(CENTER,
    compose(
      pin(CENTER_RIGHT, left),
      pin(CENTER_LEFT, right)
    )
  );
}
```



Using `compose` and `pin`, write your own version of the above and `overlay` methods:

```
public static Graphic myAbove(
```

```
public static Graphic myOverlay(
```

How general is `compose`? Let's determine what `compose(w, t)` can produce with differently pinned arguments, assuming `w` is a red wide 80-by-32 rectangle, and `t` is a blue tall 32-by-80 rectangle. Sketch some of the 81 possible results:

<code>w</code> →									
<code>t</code>	TL	TC	TR	CL	C	CR	BL	BC	BR