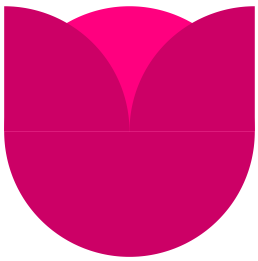
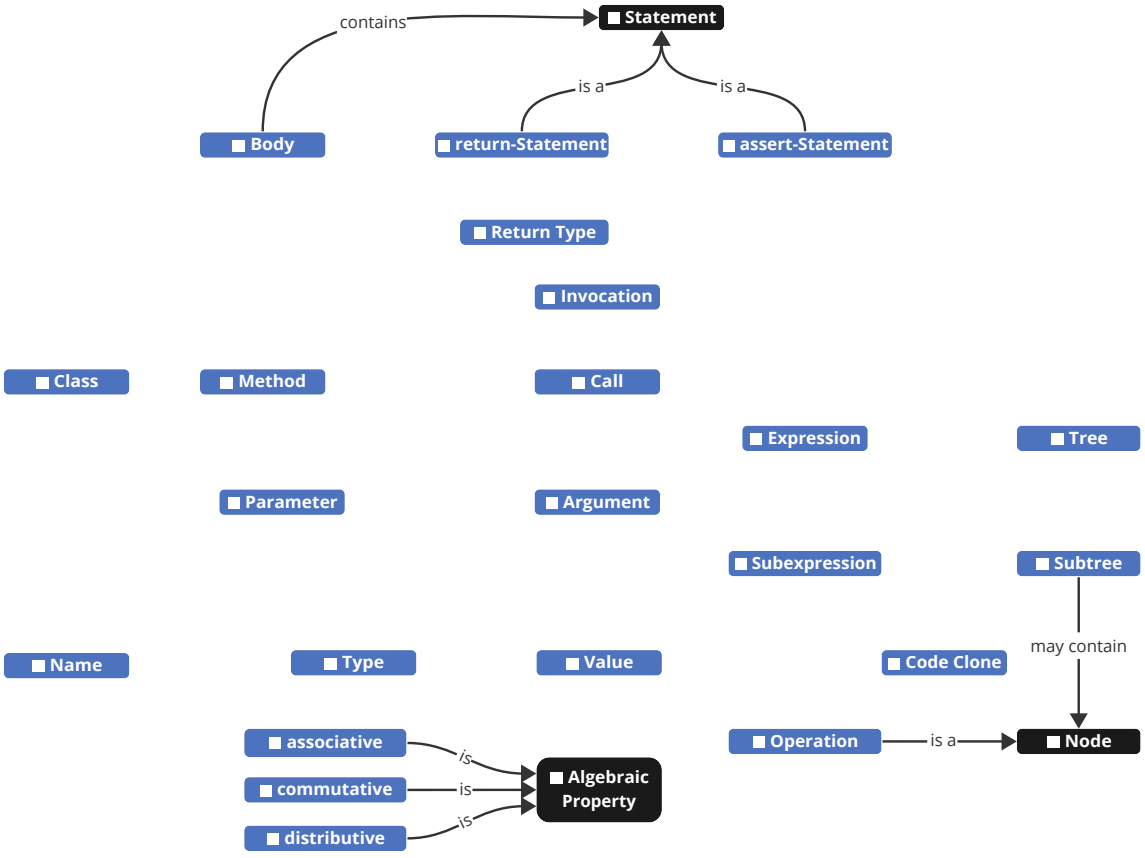
Methods, Expressions, Parameters

Student name:	TA signature:



Concepts Check off understood concepts, connect related concepts, label connections



Make sure you can **explain** each concept and each connection, you can provide **examples**, and you can **identify** them in a given piece of code.

Names Circle the methods, underline the types

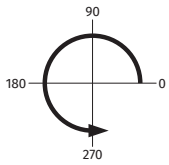
`rectangle` • `triangle` • `ellipse` • `circularSector` • `text` • `Graphic` • `Color` • `RED` • `GREEN` • `BLUE` • `CYAN` • `MAGENTA` • `YELLOW` • `BLACK` • `WHITE` • `rotate` • `overlay` • `above` • `beside` • `Toolbelt`

Calling Methods - Primitive Graphics

JTamaro, the library we use in this course, provides methods to produce graphics. Here are the headers (return type, name, parameters) of these methods:

```
Graphic rectangle(double width, double height, Color color)

Graphic triangle(double side1, double side2, double angle, Color color)

Graphic ellipse(double width, double height, Color color)

Graphic circularSector(double radius, double angle, Color color)

Graphic text(String content, String font, double fontSize, Color color)
```

The `angle` parameter in the `triangle` and `circularSector` methods (like angles in every other part of JTamaro) is specified in **degrees**, with 0 degrees pointing right, and positive angles going counterclockwise.



JTamaro also provides some names for various values of type `Color`:

```
RED   GREEN   BLUE   CYAN   MAGENTA   YELLOW   BLACK   WHITE
```

In Java, to call (invoke) a method, we write the name of the method, followed by parentheses that may contain arguments (one argument for each parameter of the method):

```
circularSector(10, 90, RED)
```

Write expressions that invoke the methods documented above to produce...

a red **rectangle** with width 200 and height 100:

a blue **square** with side length 200:

a yellow **equilateral triangle** with side length 200:

a black **circle** with diameter 200:

a magenta **text** with the content “Ciao!” in Helvetica font of size 200:

Nesting Method Calls - Rotation of a Graphic

JTamaro provides a method that produces a rotated (counterclockwise) copy of a given graphic:

```
Graphic rotate(double angle, Graphic graphic)
```

To call this method, you need to provide two arguments: an angle and a graphic:

```
rotate(90, ellipse(200, 100, RED))
```

Write expressions that invoke `rotate` and other methods to produce...

a black **diamond** (square standing on its corner) with side length 200:

a **text** "Wrong way!" rotated by 180 degrees in red 200-size Helvetica:

a **text** "upwards" slanted 30 degrees upward in red 200-size Helvetica:

Combining Two Graphics: overlay, beside, and above

JTamaro provides a method that produces a new graphic that consists of one graphic overlaid on top of the center of another graphic:

```
Graphic overlay(Graphic top, Graphic bottom)
```

To call this method, you need to provide two arguments of type `Graphic`:

```
overlay(rectangle(10, 10, WHITE), rectangle(200, 200, BLACK))
```

Write expressions that invoke `overlay` and other methods to produce...

a black **circle** with diameter 200 on top of a white **square** with side length 200:

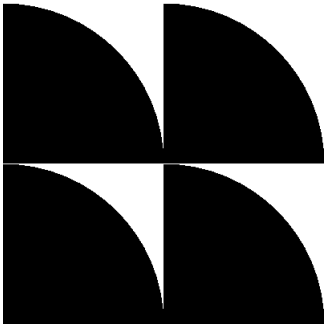
a black **circle** with diameter 100 on top of a white **circle** with diameter 200:

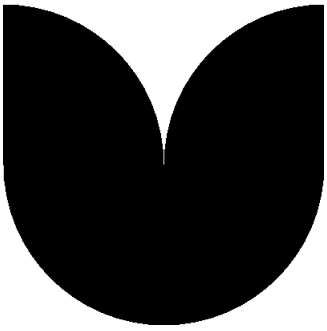
JTamaro also provides two other methods that combine two graphics:

```
Graphic above(Graphic top, Graphic bottom)

Graphic beside(Graphic left, Graphic right)
```

Write expressions that invoke `above`, `beside`, and other methods to produce...

```
beside(
  circularSector(200, 90, BLACK),
  circularSector(200, 90, BLACK)
)
```





Seeing Expressions as Trees

Expressions are pieces of source code that produce a value. Here is an example:

```
overlay(rectangle(10, 10, WHITE), rectangle(200, 200, BLACK))
```

We can write the same expression differently, with extra spaces and line breaks:

```
overlay(
  rectangle(10, 10, WHITE),
  rectangle(200, 200, BLACK)
)
```

Other formats are possible. Which of the following do you prefer? Why?

(a)	(b)	(c)

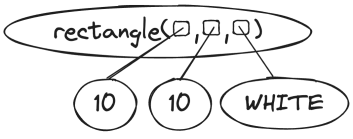
`overlay(`
`  rectangle(`
`  10,`
`  10,`
`  WHITE`
`  ),`
`  rectangle(`
`  200,`
`  200,`
`  BLACK`
`  )`
`)`
	`overlay`
	`  (`
	`rectangle (  10 ,`
	`  10`
	`,WHITE`
	`  )`
	`  ,rectangle(200,200`
	`,BLACK`
	`  )`
	`  )`
		`overlay(`
		`  rectangle(`
		`  10,`
		`  10,`
		`  WHITE`
		`),`
		`  rectangle(`
		`  200,`
		`  200,`
		`  BLACK`
		`)`
		`)`

Format (a) consistently brings out the **nested hierarchical structure** of the expression. An expression forms a tree, where each subtree corresponds to a subexpression (an expression inside another expression).

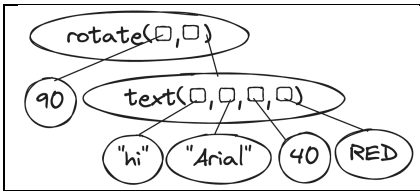
Which of the following code pieces are expressions (in Java)? Why/why not?

- ☐ `overlay(`
- ☐ `10`
- ☐ `,`
- ☐ `200, 200`
- ☐ `(10, 10, WHITE)`
- ☐ `WHITE`
- ☐ `rectangle(10, 10, WHITE)`
- ☐ `rectangle`

Draw the tree for each of the following expressions:

Code:	`200`	`rectangle(10, 10, WHITE)`	`WHITE`	`circleOnSquare()`
Tree:				

Write the expression corresponding to the given tree:

Methods – Named, Reusable Expressions

Remember the expression to create a circle on a square?

```
overlay(
    ellipse(100, 100, BLACK),
    rectangle(100, 100, WHITE)
)
```

Every time you need a circle on a square you have to write all that code. Wouldn't it be nice if you could write the code **only once**, give it a name, and then just write that name whenever you need a circle on a square?

In Java that's possible. You can **create your own method**, with a name of your choice, put the expression inside that method, and call the method whenever you want to evaluate the expression. Here is how you define such a method:

```
    public static Graphic circleOnSquare() {
        return overlay(
            ellipse(100, 100, BLACK),
            rectangle(100, 100, WHITE)
        );
    }
```

The part inside the curly braces { } is the **method body**. It consists of a **return statement**. That return statement contains an expression (highlighted) that produces the value the method will return. The word `Graphic` in front of the method name defines that this method will return a value of type `Graphic` whenever we call it. We will explain the `public` and `static` keywords in the future. For now, make all your methods `public` and `static`.

Now that you have that method, you can create a circle on a square by calling it:

```
circleOnSquare()
```

Package the following two expressions into methods with the given names:

Expression	Method

`overlay(`
`  ellipse(100, 100, BLACK),`
`  ellipse(200, 200, WHITE)`
`)`
	`circleOnCircle`

`overlay(`
`  ellipse(100, 100, BLACK),`
`  ellipse(200, 100, WHITE)`
`)`
	`circleOnEllipse`

Reducing Code Duplication

The `circleOnSquare`, `circleOnCircle`, and `circleOnEllipse` methods share at least one common **subexpression** – a part of the expression in their return statements is the same. A subexpression is an expression that sits inside another expression. Write down the biggest subexpression that's common in the three methods:

If we have to write the same code more than once, we call that a **code clone**. Creating code clones is **bad**. If we have multiple copies of the same code, it becomes difficult to keep them in sync when we need to change them in some way (e.g., here, if we'd like to change the color from `BLACK` to `RED`, we'd have to do that in **three** different places).

When we see a code clone, we should very seriously consider **eliminating** it. How? By packaging the clone into a method, and by then simply calling that method wherever we previously had a clone. This is the "extract method" refactoring.

Let's extract the common subexpression from above into a separate method, named `blackCircle`. Write that method:

`blackCircle`

Now, let's rewrite the three methods, and, in their bodies, replace the clones with calls to `blackCircle`:

Code with Clones	**Clones Extracted into `blackCircle` Method**

`public static Graphic circleOnSquare() {`
`  return overlay(`
`  ellipse(100, 100, BLACK),`
`  rectangle(100, 100, WHITE)`
`  );`
`}`
	`public static Graphic circleOnSquare() {`

`public static Graphic circleOnCircle() {`
`  return overlay(`
`  ellipse(100, 100, BLACK),`
`  ellipse(200, 200, WHITE)`
`  );`
`}`
	`public static Graphic circleOnCircle() {`

`public static Graphic circleOnEllipse() {`
`  return overlay(`
`  ellipse(100, 100, BLACK),`
`  ellipse(200, 100, WHITE)`
`  );`
`}`
	`public static Graphic circleOnEllipse() {`

Reducing Code Duplication – Another Example

Reduce code duplication in the following code (cross out and replace code right here in the listing). You can introduce new methods.

```
public static Graphic besideSame() {
  return beside(
    circularSector(200, 90, BLACK),
    circularSector(200, 90, BLACK)
  );
}

public static Graphic besideRotated() {
  return beside(
    rotate(0, circularSector(200, 90, BLACK)),
    rotate(90, circularSector(200, 90, BLACK))
  );
}

public static Graphic twoByTwoSame() {
  return above(
    beside(
      circularSector(200, 90, BLACK),
      circularSector(200, 90, BLACK)
    ),
    beside(
      circularSector(200, 90, BLACK),
      circularSector(200, 90, BLACK)
    )
  );
}

public static Graphic twoByTwoRotated() {
  return above(
    beside(
      circularSector(200, 90, BLACK),
      rotate(90, circularSector(200, 90, BLACK))
    ),
    beside(
      rotate(180, circularSector(200, 90, BLACK)),
      rotate(270, circularSector(200, 90, BLACK))
    )
  );
}
```

Operations Can Satisfy Some Properties

Assume you have the following two methods:

```
public static Graphic wide() {
  return rectangle(200, 80, RED);
}

public static Graphic tall() {
  return rectangle(80, 200, BLUE);
}
```

Write code that makes one call to `wide`, one call to `tall`, and one call to either `above`, `beside`, or `overlay`. What are all the different graphics you can produce?

Code	Sketch of Graphic
`above(wide(), tall())`	

Is the graphic produced by `above(wide(), tall())` equal to the graphic produced by `above(tall(), wide())`?

☐ yes ☐ no

Is the number produced by `1 + 2` equal to the number produced by `2 + 1`?

☐ yes ☐ no

Is the number produced by `1 - 2` equal to the number produced by `2 - 1`?

☐ yes ☐ no

Which property does a binary operation $\oplus$ (a method with two parameters, an operator with two operands) where `a` $\oplus$ `b = b` $\oplus$ `a` satisfy?

◯ associativity ◯ commutativity ◯ distributivity

As your sketches show, neither `above`, nor `beside`, nor `overlay` is commutative.

Parameterize

Existing methods like `rectangle` have **parameters**. To call these methods, you have to provide a list of **arguments**, one argument for each parameter.

The methods we **developed** ourselves so far have not had any parameters. Let's create some methods with parameters:

A method that creates a red **rectangle** with the width corresponding to the value of a parameter, and the height corresponding to half of that parameter's value:

```
public static Graphic redRectangle(double size) {
  assert size >= 0;
  return rectangle(size, size / 2, RED);
}
```

Notice that we specify a meaningful name for the parameter (`size`), that we have to specify its type (`double`, i.e., a floating-point number), and that we check that the parameter's value is acceptable (**assert**, i.e., it won't accept negative sizes).

A method that creates a blue **square** with side length given as parameter:

A method that creates a yellow **isosceles triangle** with the length of the two equally long sides given as parameter and an angle of 30 degrees:

A method that creates a black **ellipse** with height corresponding to the value of a parameter, and the width corresponding to twice of that parameter's value:

A method that creates a magenta **text** with the content "Ciao!" in Helvetica font of a size given as a parameter:

Let's Build a Toolbelt of Reusable Methods

Whenever we have to write the same expression over and over again, we better put that expression inside a method. Throughout the remainder of this course, we will **collect** such reusable methods in a class we call `Toolbelt`. A class is a nice way to group together several related methods.

Throughout this workbook you had to create circles several times. You had to call the `ellipse` method for that, which is complicated. You did create a method named `blackCircle`, which draws a black circle with a diameter of 100. But we will need circles with different diameters and different colors.

Let's write a more general method for creating a **circle** of any given diameter and any given color:

```
public static Graphic circle(double diameter, Color color) {
  assert diameter >= 0;


}
```

Let's do the same for a method that creates a **square**. Come up with good names for the parameters:

```
public static
  assert
```

From now on, whenever we have a situation where we need to write similar pieces of code several times, we will create a method instead. If the method seems generally useful, we will place it in our `Toolbelt` class, so that we know where to find it in the future. Over time, we will collect more and more reusable tools (methods) in our belt, and we can just call them instead of creating unnecessary code clones.

Using your Toolbelt

If you now want to create a circle, you can conveniently call the `circle` method. However, if the code that tries to call the `circle` method is not itself *inside* the `Toolbelt` class, then you have to say that you want to call the `circle` method of the `Toolbelt` class. You do that by writing the class name, followed by a dot, followed by the method:

```
Toolbelt.circle(300, RED)
```

From another class, use your toolbelt's `square` method to create a red **diamond** (i.e., a square rotated by 45 degrees) with a side length of 200:

From another class, use your toolbelt's methods to create a black **square** with side length 200 beside a green **circle** with diameter 200: