Access specifiers: public and private data and methods

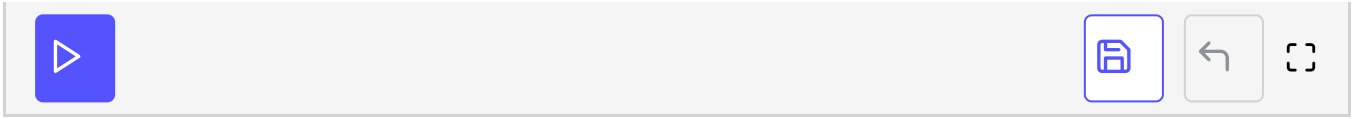
Learn how to use the keywords public and private, and how to use getter and setter methods to maintain consistent object state.

WE'LL COVER THE FOLLOWING

Instance variables of an object can be accessed using dot notation, just like in Python, Javascript, or C++. Here's the simple example of defining a `Circle` class, which allows `Circle` objects to be created. Each circle object has the instance variables `x`, `y`, and `r`:

```
// include educative's simple graphics library:
import com.educative.graphics.*;

class Circle {
  public int x;
  public int y;
  public int r;
}

class CircleTest {
  public static void main(String[] args) {
    Circle circ;
    circ = new Circle();

    // set values for the instance variables for the circle:
    circ.x = 100;
    circ.y = 100;
```

```
    circ.r = 50;

    Canvas c = new Canvas(200, 200);

    c.fill("yellow");
    c.stroke("black");

    c.circle(circ.x, circ.y, circ.r);
  }
}
```



We can access the radius of the circle with `circ.r` in Java, just like in Python or Javascript, and we've done so in the above code.

However, in Java, it is considered **bad form to directly access instance variables** from methods outside the class. Why?

1. Internal representations change. If I decide later that the radius of the circle should be a double, and that the variable name should be `radius`, I might remove the instance variable `r`, breaking every piece of code that depends on it.

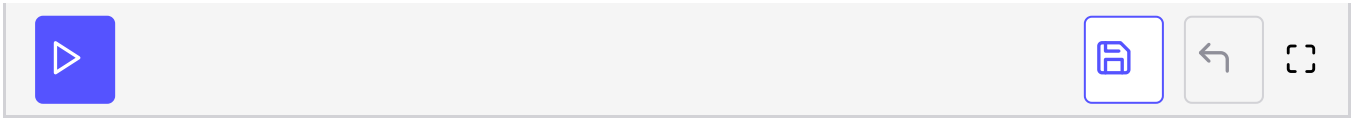
2. No error checking is performed when modifying an instance variable directly. For example, if I write the code `circ.r = -50`, then the circle is in an inconsistent state.

Setter methods

If we are not allowed to set the value of `r` directly from outside the `Circle` class, how should we change the radius of the circle? One way of modifying instance variables is a **setter** method. Such a method may be called from outside the class, and handles the management of instance variables as needed. For example:

```
// include educative's simple graphics library:
import com.educative.graphics.*;

class Circle {
  public int x;
  public int y;
  public int r;

  public void setRadius(int radius) {
    if(radius < 0) {
      throw new IllegalArgumentException("Circle radius must be non-negative.");
```

```
    }
    this.r = radius;
  }

}

class CircleTest {
  public static void main(String[] args) {
    Circle circ;
    circ = new Circle();

    // set values for the instance variables for the circle:
    circ.x = 100;
    circ.y = 100;
    circ.setRadius(20);

    Canvas c = new Canvas(200, 200);
    c.fill("yellow");
    c.stroke("black");

    c.circle(circ.x, circ.y, circ.r);
  }
}
```



Run the above code. Then change the argument in `circ.setRadius(20)` to -20, and run the code again. You'll see that an exception is generated and the code crashes. Good! Now the programmer knows that there is a bug and can fix it.

To generate an error message, the method `setRadius` makes use of a Java command called `throw`. The keyword `throw` generates an error which, if not handled, causes the program to crash. We will not go into details of how `throw` (and its sister `catch`) work, but once you are more comfortable with the basics of Java, exception handling is a useful topic to explore.

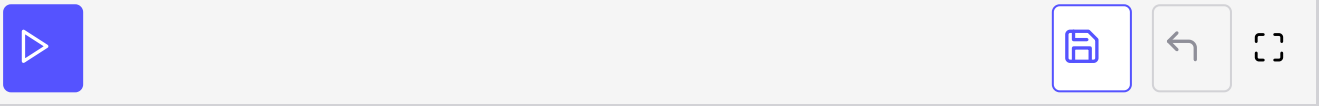
Exercise: `setLocation` method

In the code below, write a method `setLocation` that takes two parameters, an x and y coordinate, and sets the instance variables `x` and `y` of the circle to those coordinates. You do not need to do any error checking, since coordinates may legally have negative values. Also write a constructor that lets the initial values be set when the circle is created.

CircleExample.java | Sample solution

```
// include educative's simple graphics library:
```

```
import com.educative.graphics.*;

class Circle {

  public int x;
  public int y;
  public int r;


  // Write your constructor here:

  // Write your setLocation method here:

  public void setRadius(int radius) {
    if(radius < 0) {
      throw new IllegalArgumentException("Circle radius must be non-negative.");
    }
    this.r = radius;
  }

}

class CircleExample {
  public static void main(String[] args) {

    Canvas c = new Canvas(200, 200);
    c.fill("yellow");
    c.stroke("black");

    Circle circ;
    circ = new Circle(100, 50, 40);
    c.circle(circ.x, circ.y, circ.r);


    circ.setLocation(100, 125);
    circ.setRadius(20);

    c.circle(circ.x, circ.y, circ.r);


  }
}
```



In the sample solution, I called the methods `setLocation` and `setRadius` from the constructor. Although this is not required, it's nice to make use of the error checking in `setRadius` when setting the initial radius using the constructor, and if I later add new behavior to `setLocation`, then the constructor will make use of it automatically.

The keyword `private`

The *principle of least privilege* suggests that we should give code the absolute

minimum capabilites to do the needed job. Direct access to variables is a

powerful capability, and often not needed. I do not need the capability of changing the radius of a circle to be a negative number.

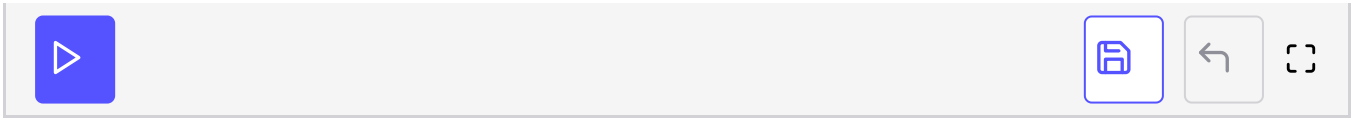
Now that we have methods to set the location and radius of the circle, we don't ever need to write lines of code like `circ.r = 20`. In fact, we'd like to force programmers to use `setRadius` rather than accessing the instance variable directly, so that we can make sure that error checking is always performed.

So far, we have always made instance variables `public`. This means that any code can access those variables with a reference to the object and dot notation. `public` is called an *access specifier*.If we change the *access specifier* to `private`, then only code within the `Circle` class can access the instance variable. `circ.r = 20` is an illegal instruction if `r` is a private instance variable.

Here's the code with the access specifiers changed to private:

```
// include educative's simple graphics library:
import com.educative.graphics.*;

class Circle {
  private int x;
  private int y;
  private int r;

  public Circle(int x, int y, int r) {
    this.setLocation(x, y);
    this.setRadius(r);
  }

  public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void setRadius(int radius) {
    if(radius < 0) {
      throw new IllegalArgumentException("Circle radius must be non-negative.");
    }
    this.r = radius;
  }

}

class CircleExample {
  public static void main(String[] args) {
```

```
    Canvas c = new Canvas(200, 200);
    c.fill("yellow");
    c.stroke("black");


    Circle circ;
    circ = new Circle(100, 50, 40);
    c.circle(circ.x, circ.y, circ.r);

    circ.setLocation(100, 125);
    circ.setRadius(20);

    c.circle(circ.x, circ.y, circ.r);

  }
}
```



Run the above code. Notice that the compiler gives you some error messages. Specifically, there is something wrong with the line of code `c.circle(circ.x, circ.y, circ.r);`.

Aha! We used the `private` access specifier: we cannot modify or even use the value of the instance variables `x`, `y`, or `r` from outside the class.

One way to handle this problem would be to write a method `public void draw` within the `Circle` class, like we did for the `Ball` class in the previous lesson. The `draw` method is inside the `Ball` class and has access to the instance variables of the object referred to by `this`. In general, the `this` keyword is the only way to access a private instance variable. (You may eventually omit the keyword `this`, leaving it implied, but for now, you should use it.)

Try writing a draw method and using it to replace the code `c.circle(circ.x, circ.y, circ.r);`. Make sure your draw method accepts a reference to a Canvas object, since it will need it to call `circle`.

Getter methods

What if you really want to know the radius of some Circle object from outside of the `Circle` class? If the instance variables are `private`, and they should be, `circ.r` is unavailable. One way to handle this is to write a method `getRadius` that returns the value of the radius. Such a method is called a **getter**. Let's do it:

Exercise: methods `getX`, `getY`, `getRadius`

Write getter methods that get the values of `x`, `y`, and `r` instance variables. Notice that each of these methods returns an `int`, so make sure to specify that in the method header. The body of each method should be only one line long.

CircleExample.java | Sample solution

```
// include educative's simple graphics library:
import com.educative.graphics.*;

class Circle {
  private int x;
  private int y;
  private int r;

  public Circle(int x, int y, int r) {
    this.setLocation(x, y);
    this.setRadius(r);
  }

  public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public void setRadius(int radius) {
    if(radius < 0) {
      throw new IllegalArgumentException("Circle radius must be non-negative.");
    }
    this.r = radius;
  }

  // write your getter methods here:

}

class CircleExample {
  public static void main(String[] args) {

    Canvas c = new Canvas(200, 200);
    c.fill("yellow");
    c.stroke("black");

    Circle circ;
    circ = new Circle(100, 50, 40);

    System.out.println("The radius of circ is " + circ.getRadius());

  }
}
```

Writing getter methods may seem annoying, but you do not need to write a getter method for every instance variable: just for the ones you want direct

access to. And in fact, getter methods have an additional features: they can separate internal representation of data from the external *view* of that data.

For example, perhaps you have written a thermometer class. Internally, you might represent the temperature using the Kelvin scale. You could write getter methods `getCelsius` and `getFahrenheit` to do the conversions and access the data.

Most instance variables should be private

Almost all instance variables should be `private`. This ensures that access to instance variables is done in a controlled fashion, and creates nice barriers between different modules of code. If an instance variable is `private`, you can debug the class you are working on secure in the knowldege that the instance variable's value was set somewhere within the current class.

The only time you might have a public instance variable is for a very small class that packages simple values together. For example, the class `java.awt.geom.Point2D` provided by Java has public `x` and `y` instance variables. These variables represent the integer coordinates of a point and are not restricted in any way.

Some methods are public, and others are private

So far, we have only seen public methods. Getter and setter methods must be public, of course, and so must constructors.

Sometimes it is useful to factor a method into smaller methods, however, and those smaller methods may not have usefulness beyond the current small piece of code you are working on. In this case, these methods may be declared `private`.

Think hard about which methods you should make public

In general, you should aim to have only a few public methods for each class: these methods make up the public, documented interface for using the class.

Once you have created a public method, other people may use that method, and you must always make sure that as your code evolves, that method is still

and you must always make sure that as your code evolves, that method is still

available. **If you change the name, parameters, or behavior of a public method, you will break all code that makes use of that method.**