Classes define objects

A class is used to define the custom data structure that makes up an object of that class.

WE'LL COVER THE FOLLOWING

We have seen that classes are the organizational unit of a Java program. Source code for methods is always within a class, and we might expect related methods to be in the same class.

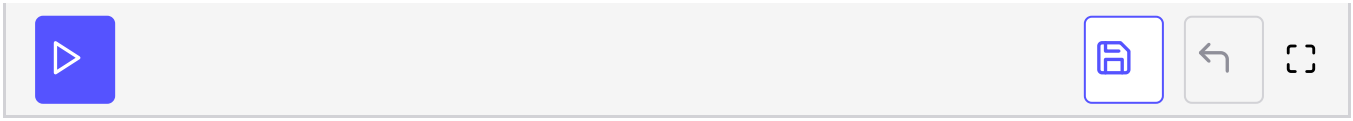
However, classes do much more than this, and are the heart of object-oriented programming in Java:

1. A class is used to define the **custom data structure** that makes up an object.
2. Classes define **methods** that act on those objects.

Here's an example of creating a custom data type, `Circle`, to store information about a circle:

```
// 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;   // declare a variable to store a reference to Circle object

    // create a new Circle object in memory, and store a reference to it:

    circ = new Circle();

    // set values for the instance variables for the circle:
    circ.x = 100;
    circ.y = 100;
    circ.r = 50;

    // Create a Canvas object for drawing on the screen:
    Canvas c = new Canvas(200, 200);
    c.fill("yellow");
    c.stroke("black");

    // use instance variables of the the circle object when calling
    //  the circle method of the canvas (used to draw the circle)
    c.circle(circc.x, circ.y, circ.r);
  }

}
```



Notice that there are two different classes defined in this file. Typically in Java, each class would get its own file, with the name of the file `ClassName.java`. So `Circle` might be defined in a class `Circle.java`. This is a good convention and we should follow it; the classes are in the same file here only because it is easier to read on the web page.

A class declares what instance variables an object of that class will contain

Just like in Javascript, Python, or C++, an object is a data structure stored on the heap area of memory, containing data elements. These elements are called **instance variables** in Java. A class definition lists out the types and names that each object of that class will have. So each `Circle` object will contain three `int` values: `x`, `y`, and `r.`.

In Python or Javascript, instance variables of an object are created using a special constructor method. Java also has constructors, but in Java, variables must be declared before they can be used or assigned values to – even the constructor cannot create a new variable that has not been explicitly declared.

In Java, **every instance variable must be declared in the class.** Notice that the declarations for the instance variables occur **outside** of any method in `class Circle`: the class just lists them out. That's because variables declared inside of method definitions are local variables, and are destroyed once the method returns.

Also notice the keyword **public** before each of these instance variables. This indicates that methods of other classes can access these variables. We'll see more later about the difference between public and private variables or methods.

Creating an object of a class

Although a class describes the instance variables of an object, no instance variables exist until an object of the class is created. The command `new`, which must be followed by a call to the constructor of a class, creates space on the heap for the object, and returns a reference to that object:

```
Circle circ;
circ = new Circle();
```

You can think of the reference as the address of the object in memory. So the above code creates a new object on the heap with the code `new Circle()`, and stores a referene to that object in the variable `circ`. Like every other variable in Java, `circ` must have a type. The declaration `Circle circ` creates the variable and indicates that it will store a reference to an object of type `Circle`.

You might notice that there is no definition for a function or method `Circle()` anywhere in the code. Java creates a *default constructor* function for you automatically, with the same name as the name of the class.

Constructors

Object-oriented design drives the structure of the Java language. Classes and objects are intended to model specialized types of data; in this case, a circle. When you first create an object using the default constructor, space in

memory is allocated for the variables, but they are not assigned any values.

You almost always want to write your own constructor, which will let you give initial values to instance variables. Here's an example:

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

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

  public Circle(int initX, int initY, int initR) {
    this.x = initX;
    this.y = initY;
    this.r = initR;
  }
}

class CircleTest {
  public static void main(String[] args) {
    Circle circ;   // declare a variable to store a reference to Circle object

    // create a new Circle object in memory, and store a reference to it:
    circ = new Circle(100, 100, 50);

    // Create a Canvas object for drawing on the screen:
    Canvas c = new Canvas(200, 200);
    c.fill("yellow");
    c.stroke("black");
    c.circle(circ.x, circ.y, circ.r);
  }
}
```

Some rules to remember:

1. The constructor has the same name as the class.
2. The constructor should initialize all instance variables of the object, possibly using some passed parameters.
3. The constructor has access to a reference to the newly created object, stored in the special variable `this`.

Like in other languages, constructors are a nice place to check for good values. For example, you might verify that the value of the radius for the circle is non-negative, and if not, halt the program.

The special variable `this`

Instance variables of an object are accessed using dot-notation, which requires a reference to an object, a dot, and the name of the instance variable.

The job of the constructor is to initialize the instance variables of some object. That object was just created with the command `new`, followed by a call to the constructor. In order to access the instance variables, the constructor requires a reference to that newly created object. The variable `this` is maintained by `Java`, and contains the needed reference.

The code `this.x = initX` therefore takes the value of the constructor parameter `initX`, 100 in this case, and stores it into the instance variable `x` of the object referenced by `this`.

The variable `this` is essentially identical to the variable `self` in Python, and somewhat related to the variable `this` in Javascript: each of these variables stores a reference to some object, which can be used to manipulate the instance variables of that object.

In Python, the constructor is defined by writing the body of the special method `__init__(self, ...)`, and the first parameter, `self`, is written into the parameter list. Java constructors are named using the name of the class, and the parameter `this` is not included in the parameter list, but is nonetheless available automatically.

The variable `this` is also available in non-static method calls other than the constructor, as we will see shortly, even though `this` is not an explicit parameter to those methods.

Sometimes you will see Java code that omits the `this.` when accessing instance variables. For now, I recommend that you always use `this` when accessing instance variables in the constructor: the keyword makes it extremely clear that you are accessing an instance variable, and not some local variable or parameter to the call.

Since you will see it code written by others, here's an example of omitting `this`:

```
class Circle {
```

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


  public Circle(int initX, int initY, int initR) {
    x = initX;
    y = initY;
    r = initR;
  }
}
```

I chose to name the first parameter to the constructor `initX`. Some might choose to call that parameter `x`, since it will contain the value of the `x` coordinate of the circle's center. There's nothing wrong with that choice, but if parameters and instance variables have the same name, you cannot omit the word `this.` from the assignment to the instance variable:

```
  public Circle(int x, int y, int r) {

    // Notation with "this." on the left-hand side makes it clear
    //  that we are assigning the values of parameters into instance variables.
    this.x = x;
    this.y = y;
    this.r = r;
  }
```

Exercise: Ball class

Write a class `Ball` that will be used to model a ball bouncing on the screen. Your ball should have five instance variables: a `String` describing the color of the ball, and `int` values describing the x location, a y location, and x and y components of the velocity of the ball, vx and vy. (For now, the velocities will just be set to zero. We will use them in the next lesson.)

All of the instance variables should be declared as public, so that they may be accessed by the `drawBall` method in the main `BallExample` class.

You'll need to:

1. declare the appropriate variables;
2. write a constructor.

Read the code in `main` to see how your constructor should be organized.

BallExample.java | Sample solution

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

class Ball {
  // you write this part:
}

class BallExample {
  public static void drawBall(Canvas canvas, Ball ball) {
    canvas.fill(ball.color);
    canvas.stroke("black");
    canvas.circle(ball.x, ball.y, 10);
  }

  public static void main( String args[] ) {
    Canvas c = new Canvas(200, 200);

    // create a red ball at location (20, 30) with 0
    //  x and y velocity:
    Ball b = new Ball("red", 20, 30, 0, 0);

    drawBall(c, b);
  }
}
```