//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//NOTES ON ACCELERATION AND VELOCITY-LIMIT VARIABLES:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

//\*\*\*\*\*TLDR:\*\*\*\*\*

//-If your never want your object (namely an enemy) to accelerate in the x directions,

// always have your object's accelerationX value set to 0. Your object's overridden

// Update() function will just consist of your object's AI determining which x-direction

// to move in, and then setting velocityX to a value in that direction. You needn't

// bother with velocityLimitX. If you want an enemy to run right, for example, you

// can simply set its velocityX to a positive value. If you ever want it to start moving

// in the opposite (left) direction, you just set its velocityX to a negative value.

// The PhysicsEngine class will take care of its collision into walls: upon collision,

// the PhysicsEngine class will call CollideX(), which by default resets velocityX to 0

// However, you can override this so that it multiplies velocityX by -1, thereby making

// the enemy turn around. Or you could even set velocityY to a negative value if you want

// the enemy to try and jump over whatever obstacle is in its way.

//

//-If your never want your object (namely an enemy) to accelerate in the y directions,

// always have your object's accelerationY value set to 0. Your object's overridden

// Update() function will just consist of your object's AI determining which y-direction

// to move in, and then setting velocityY to a value in that direction. You needn't

// bother with velocityLimitY. If you want an enemy to jump, for example, it's as simple

// as setting its velocityY variable to a negative value. The PhysicsEngine class will

// take care of the jump arc and descent. When your enemy touches a ground again, the

// PhysicsEngine class will call the enemy's Land() method, resetting its velocityY to 0.

//

//-If you \*do\* want acceleration in either of these directions, then it's best to

// coordinate each new assignment to the acceleration variable with an assignment of

// an appropriate value to the related velocityLimit variable. If you don't want to bother

// with that, then you can keep those variables set at high absolute values that the

// object is unlikely to reach in its environment.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//THE FULL EXPLANATION:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//The accelerations will change the above velocities each frame, stopping if and when

//the velocity limits are reached.

//

//If you want an object to accelerate in a direction,

//it's smart to assign a value to velocityLimit so that your object never goes over/

//under that limit. Otherwise, your object will move dangerously crazy-fast. Unless

//you want that, in which case you can just set velocityLimit to a very high value.

//

//To clarify, the accelerationY variable has nothing to do with the GRAVITY constant

//in the PhysicsEngine class -- accelerationY is the acceleration of the object's

//own \*intended, controlled\* motions, whereas GRAVITY is an \*external\* acceleration

//applied to a gravity-affected, ungrounded object regardless of its intended motions.

//There is no need to account for gravity with this accelerationY: that accounting will

//be done in the Update() method of the PhysicsEngine class.

//

//NOTE: These 4 variables are only necessary for objects that \*do\* accelerate/decelerate,

//such as the player's character. Some objects, such as the vast majority of enemies in

//past action or platforming games, don't accelerate or decelerate; they simply move at

//one constant horizontal magnitude, and if they collide into a formation or decide

//to turn around then their current velocity is simply assigned a new value. If the x-

//and/or y-movement of your object fits this description, you can simply keep that

//acceleration variable set to 0 at all times. The update method of that object can simply

//change its velocity values when collisions or AI requires it. If you do this, then the

//displacement for that frame will be determined entirely by the product of the velocity

//and the total elapsed seconds.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//NOTES ON THE ARRAYS collisionXs AND collisionYs

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

//\*\*\*\*\*TLDR:\*\*\*\*\*

//To Create collisionXs:

//-First element is the coordinate of the left edge of the hit box.

//-Last element is the coordinate of the right edge of the hit box.

//-If the agreed minimum object width is I, include every Ith integer

// after the left edge's value, stopping at the first integer to

// equal or exceed then right edge's value.

//

////To Create collisionYs:

//-First element is the coordinate of the top edge of the hit box.

//-Last element is the coordinate of the bottom edge of the hit box.

//-If the minimum agreed object height is J, include every Jth integer

// after the top edge's value, stopping at the first integer to

// equal or exceed then bottom edge's value.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//THE FULL EXPLANATION:

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//These arrays contain all the x- and y-coordinates, respectively, of the all object's

//collision points.

//

//A collision point will be created from each possible combination of one element in

//collisionXs and one element in collisionYs. Thus for each element in collisionXs,

//there will be a number of collision points, equal to the number of elements in

//collisionYs, whose x-coordinate is equal to that element from collisionXs. Thus if

//collisionXs has m elements and collisionYs has n elements, the object will have

//m \* n collision points.

//

//As an example, suppose collisionXs == {0, 5, 10} and collisionYs == {0, 5, 10}.

//Then the object has the collision points (0, 0), (0, 5), (0, 10), (5, 0), (5, 5),

//(5, 10), (10, 0), (10, 5), and (10, 10). Checking if the object is stuck will

//involve testing these nine points on the object's sprite.

//

//The first two absolutely necessary values in collisionXs are those corresponding to

//the left and right ends of hitbox, and the first two absolutely necessary values in

//collisionYs are the top and bottom ends of the hitbox. From these four values, the

//four corner collision points can be obtained. But there may be some more values

//required, as described below.

//

//Requiring the check of only a small handful of points is very efficient, but it has

//certain precautions. Since this method of collison tests only a handful of spread-out

//pixels on the object's sprite, it is possible for a skinny (or short) enough formation

//to sneak in between two of the x (or y) values in the array without touching either one,

//thereby causing the object to move completely through the formation with no collision

//detected. To prevent this, the programmers will agree on a minimum width and height

//for all collision-causing objects, floors, walls, ceilings and platforms in this game.

//None of these may be skinnier than the chosen width or shorter than the chosen height.

//From this we can determine a maximum horizontal space between two horizontally-adjacent

//collision points and a maximum vertical space between two vertically-adjacent collision

//points.

//

//With these maximums for horizontal and vertical space between points, we can determine

//all other values the arrays need. Take the smallest value in the array, which represents

//the left (or top) edge of the hit box, and keep adding to it the value for the minimum

//width (or height) until the you get a number is greater than or equal to the largest

//value in the array, which represents the right (or bottom) edge of the hitbox. Every

//value obtained by this repeated addition that lies between the low and high values

//must be included in the array.

//

//As an example, suppose our minimum height is 40 pixels (just as an example -- the real

//height could be greater than this), and an object whose sprite is 90 pixels tall has the

//top of its hitbox at y = 4 and the bottom of its hitbox at y = 86. Then the minimum

//necessary y-coordinates in collisionYs are 4, 44, 84 and 86. No two coordinates

//are more than 40 pixels apart, so no object or formation with a height of 40 pixels

//or more can be inside the object's hitbox without touching a collision point with one

//of these y-coordinates. Actually, the last value before the bottom of the hit box,

//84 in this case, can be decreased a little if you think it's weird to have to values

//that are only two pixels apart. You could replace 84 with something like 65, which is

//roughly equidistant from 44 and 86. The important thing is that you just don't simply

//omit that second-to-last element. If you did that, and had an array with only 4, 44,

//and 86, then any obstacle that is 40 or 41 pixels tall could sneak through the 44 and

//86 without causing a collision. Determining the values for collisionXs follows the

//same reasoning.

//

//It is important that these arrays are arranged in increasing order -- or, at the very

//least, arranged such that the smallest value is the first element in the array and

//the largest value is the last element. That way, methods that test for walls, ground,

//or ceilings can easily access the edges of the hitbox by using the first() and last()

//properties in the appropriate array. It will also make geometric collision detection

//between two objects easier, since first() and last() can be used similarly.

//

//Obviously, the collision points generated by these two arrays form a rectangular

//shape, so the assumption for now is that no object has a hitbox in the shape of

//something unusual like a rhombus. If we do get that far, then we can go back to

//the original idea of using one array of vector2's which represents all the collision

//points of the object. Then maybe we can store variables for left-most, right-most,

//top-most, and bottom-most coordinates of collision points, to simplify checking for

//ground/walls/ceilings. Or if we don't wanna do that, then we can use some functions

//which find min or max x- or y-coordinates in an array.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// NOTES ON PHYSICSOBJECT.UPDATE()

// Overriding this for a derived class of the object function, such

// as for an enemy, is pretty simple, especially if the object is

// not meant to accelerate. In that case, the acceleration variable

// should be set to 0 and kept at that value at all times.

//

// If and when your object should move or decides it will move in a

// particular direction, you just set the appropriate velocity variable

// to a value in that direction (e.g. velocityX > 0 for moving right,

// velocityX < 0 for moving left, velocityY < 0 for moving up,

// velocityY > 0 for moving down.) The only time the velocity needs

// to be set again is when it needs to change to a new value.

//

// As an example, if you want the object to jump when it is on the

// ground and some other conditions are met, you simply create an

// if statement based on those conditions, and inside the block the

// velocityY variable is set to a negative value. There is no need

// to set velocityY again until the object hits the ground, which

// \*will\* happen for all gravity-affected objects thanks to the

// PhysicsEngine class. Keeping accelerationY set at 0 will cause

// the object to move in the exact way you want it to.

//

// In the event the object collides with a wall/floor/ceiling, you

// have two options. The first is to override the CollideX(), Land(),

// and/or HitCeiling() methods so that the velocity changes to a

// more appropriate value. E.g. for an enemy with a constant x-

// velocity, you can override CollideX() so that its velocityX is

// multiplied by -1 when a wall is hit. The other option is to

// keep the current definitions of those functions and write

// code that handles collisions inside this Update() method. For

// example, you can write an if statement that tests whether

// wallOnLeft or wallOnRight is true, and if it is then you can

// multiply velocityX by -1 inside that block.

//

// If and when you \*do\* want your object to move in a direction with

// acceleration, the code is a little different. Instead of setting

// the velocity variable outright to a particular value, you must set

// the velocityLimit variable to the value you want the velocity

// variable to reach, and you must set the acceleration to the value

// that yields the acceleration you want for the object. If you want

// the object to speed up, acceleration and velocityLimit must have

// the same sign (positive or negative). If you want the object to

// slow down, acceleration and velocityLimit must have opposite

// signs. The absolute value of the acceleration depends on how

// quickly you want the object to reach this limit.

//

// As an example, if you want an object to reach a velocityX of 500

// pixels per second, you set velocitLimitX to 500 and choose a

// positive value for accelerationX. If you then want the object to

// skid to a stop after reaching this speed, you set velocityLimitX

// to 0 and accelerationX to a negative value.

//

// There is no need to calculate displacementX or displacementY. As long as

// you set the velocity, acceleration and sometimes velocityLimit to the

// correct values, then PhysicsObject.Update() will calculate those PROVIDED

// YOU CALL THE BASE DEFINITION of Update() at the very end of the definition.

// In fact:

//

// All derived objects must have "base.Update(time);" as the very last line

// in their own, overridden Update() method!!!

//

// Only the GameManager class should ever call this method!!!

//checkGround, checkWallRight, checkWallLeft, and isStuckAt are the collision detection

//functions. Their main test is to evaluate the color on the collision map of the pixels

//(to be) occupied by the object's collision points. But you may also notice that there

//are also tests to check whether certain coordinates are outside of the collision map's

//boundaries. For example, these tests prevent objects from moving off the left or right

//edges of the board. On the other hand, they do allow objects to move through the top

//edge or fall through the bottom edge of the board.