Computer Science 4

JavaScript

Introduces object literals, for loops, function definitions, drawing, and modulo.

#1. Dust

Level Overview and Solutions

Intro

`while` loops can use any boolean condition, not just `true` , like this:

```
while(attacks < 10) {
    hero.attack(enemy);
    attacks += 1;
}
```

The loop will end when the condition is false.

Default Code

```
// Use a while to loop until you have counted 10 attacks.

var attacks = 0;
while (attacks < 10) {
    // Attack the nearest enemy!

    // Incrementing means to increase by 1.
    // Increment the attacks variable.
    attacks += 1;
}
// When you're done, retreat to the ambush point.
hero.say("I should retreat!"); //∆ Don't just stand there blabbering!
```

Overview

Like an `if` statement, a **while-condition** loop specifies a **condition**. Every time the loop re-starts at the beginning, it checks to see if the **condition** is **true**. If so, it will execute again. If not, it stops and your program moves on to the code after the loop.

In other words, *"While **condition** is true, keep looping."*

Note that it is possible to create an **infinite loop** if your loop condition is never false!

In this level, you want to make 10 attacks, then retreat to the X mark.

To start, your `attacks` counter is `0` .

Then, create a `while` loop with the condition of `attacks < 10` .

Inside the `while` loop, find the nearest enemy, if there is one then attack it, **and increment attacks by 1**.

After the `while` loop (outside of it!), use `moveXY` to go to the X mark at coordinates: 79, 33.

Dust Solution

```
// Use a while to loop until you have counted 10 attacks.

var attacks = 0;
while (attacks < 10) {
    // Attack the nearest enemy!
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        hero.attack(enemy);
    }
    // Incrementing means to increase by 1.
    // Increment the attacks variable.
    attacks += 1;
}
// When you're done, retreat to the ambush point.
hero.moveXY(79, 33);
```

#2. Double Cheek

Level Overview and Solutions

Intro

First defeat 6 ogres in the abonded village. Then collect at least 30 gold by picking up coins that are scattered around the oasis.

A `while-loop` is a good way to repeat actions until you reach your goals.

A `while-loop` will continue looping WHILE the condition is TRUE:

```
// while the hero has less than 30 gold... collect coins.
while(hero.gold < 30) {
    var item = hero.findNearestItem();
    if(item) {
        hero.moveXY(item.pos.x, item.pos.y);
    }
}
```

Make sure you are always doing something in a `while-loop` , otherwise you may get an infinite loop error.

Default Code

```
// First, defeat 6 ogres.
// Then collect coins until you have 30 gold.

// This variable is used for counting ogres.
var defeatedOgres = 0;

// This loop is executed while defeatedOgres is less than 6.
while (defeatedOgres < 6) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
        defeatedOgres += 1;
    } else {
        hero.say("Ogres!");
    }
}

// Move to the right side of the map.
hero.moveXY(49, 36);

// This loop is executed while you have less than 30 gold.
while (hero.gold < 30) {
    // Find and collect coins.

    // Remove this say() message.
    hero.say("I should gather coins!");
}

// Move to the exit.
hero.moveXY(76, 32);
```

Overview

Like an `if` statement, a **while-condition** loop specifies a **condition**. Every time the loop re-starts at the beginning, it checks to see if the **condition** is **true**. If so, it will execute again. If not, it stops and your program moves on to the code after the loop.

In other words, *"While **condition** is true, keep looping."*

Note that it is possible to create an **infinite loop** if your loop condition is never false!

```
var defeatedOgres = 0;
while(defeatedOgres < 6) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        hero.attack(enemy);
        defeatedOgres += 1;
    }
}
```

Double Cheek Solution

```
// First, defeat 6 ogres.
// Then collect coins until you have 30 gold.

// This variable is used for counting ogres.
var defeatedOgres = 0;

// This loop is executed while defeatedOgres is less than 6.
while (defeatedOgres < 6) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
        defeatedOgres += 1;
    } else {
        hero.say("Ogres!");
    }
}

// Move to the right side of the map.
hero.moveXY(49, 36);

// This loop is executed while you have less than 30 gold.
while (hero.gold < 30) {
    // Find and collect coins.
    var item = hero.findNearestItem();
    if(item) {
        hero.moveXY(item.pos.x, item.pos.y);
    }
}

// Move to the exit.
hero.moveXY(76, 32);
```

#3. Canyon of Storms

Level Overview and Solutions

Intro

A storm is coming, but you still have a little time to collect coins before it arrives. Observe the sand yaks, they know when a storm starts.

You can use a variable as a condition for a `while-loop` , just be sure to update it inside the loop!

```
// Initialize the condition:
var yak = hero.findNearestEnemy();
while(yak) {
    // Do stuff...

    // Update the loop condition:
    yak = hero.findNearestEnemy();
}
```

Default Code

```
// A desert storm is coming!
// Yaks can detect when a storm is coming.

// This variable will be used as a condition.
var yak = hero.findNearestEnemy();

// While there is at least one sand yak:
while (yak) {
    var item = hero.findNearestItem();
    if (item) {
        hero.moveXY(item.pos.x, item.pos.y);
    }
    // Update the value of the variable yak
    // with findNearestEnemy()

}
// The yaks have hidden.
// Move to the red X at the hideout.
```

Overview

Just like we use `enemy` as the condition for an `if` statement to determine the existence of an enemy:

```
var enemy = hero.findNearestEnemy();
if(enemy) {
    hero.attack(enemy);
}
```

We can also use `enemy` (or `yak` , in this level) as the condition for a `while` loop:

```
var enemy = hero.findNearestEnemy();
while(enemy) {
    // Do something while there is an enemy
    // Be sure to update enemy inside the loop!
    enemy = hero.findNearestEnemy()
}
```

Canyon of Storms Solution

```
// A desert storm is coming!
// Yaks can detect when a storm is coming.

// This variable will be used as a condition.
var yak = hero.findNearestEnemy();

// While there is at least one sand yak:
while (yak) {
    var item = hero.findNearestItem();
    if (item) {
        hero.moveXY(item.pos.x, item.pos.y);
    }
    // Update the value of the variable yak
    // with findNearestEnemy()
    yak = hero.findNearestEnemy();
}
// The yaks have hidden.
// Move to the red X at the hideout.
hero.moveXY(38, 58);
```

#4. No Pain No Gain

Level Overview and Solutions

Intro

Time for physical training! Each second you run, the hot sun drains your health.

Change the `while` condition so that your hero runs while `hero.health` is greater than `200` .

Default Code

```
// Change the while condition.
// Run while hero's health is greater than 200:
while (true) { // Δ Change this line!
    hero.moveXY(48, 24);
    hero.moveXY(16, 24);
}

// Move to Okar.
hero.moveXY(32, 40);
```

Overview

The sample code gives you the familiar `while-true` loop, which runs forever.

In the hot sun, your hero can't run forever without passing out!

Instead, use a `while` loop with a condition so that the hero will only run `while` their `hero.health` is greater than `200` .

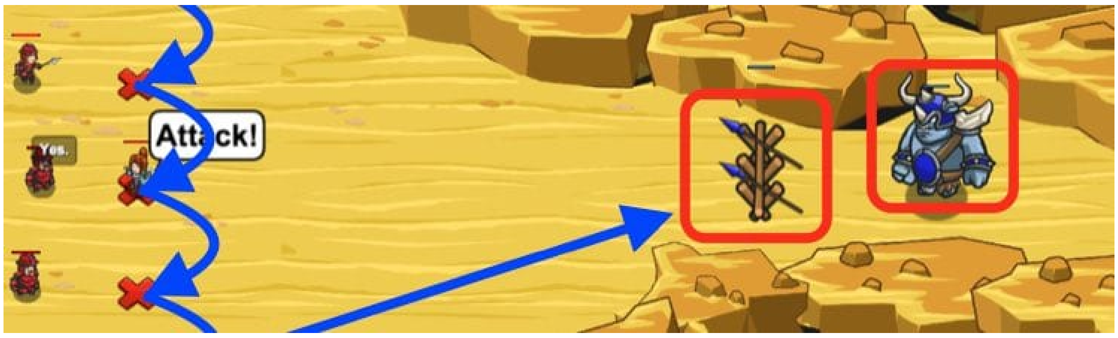
No Pain No Gain Solution

```
// Change the while condition.

// Run while hero's health is greater than 200:
while (hero.health > 200) { // Δ Change this line!
    hero.moveXY(48, 24);
    hero.moveXY(16, 24);
}
// Move to Okar.
hero.moveXY(32, 40);
```

#5. Desert Combat

Level Overview and Solutions

Intro



Be sure to increment your `while` loop condition or you will get stuck in an infinite loop!

```
var ordersGiven = 0;
while(ordersGiven < 5) {
    // If you don't do this, the loop will never end!
    ordersGiven += 1
}
```

Default Code

```
// while-loops repeat until the condition is false.

var ordersGiven = 0;
while (ordersGiven < 5) {
    // Move down 10 meters.

    // Order your ally to "Attack!" with hero.say
    // They can only hear you if you are on the X.
    hero.say("Attack!");

    // Be sure to increment ordersGiven!

}

while(true) {
    var enemy = hero.findNearestEnemy();
    // When you're done giving orders, join the attack.

}
```

Overview

Like an `if` statement, a **while-condition** loop specifies a **condition**. Every time the loop re-starts at the beginning, it checks to see if the **condition** is **true**. If so, it will execute again. If not, it stops and your program moves on to the code after the loop.

In other words, *"While **condition** is true, keep looping."*

Note that it is possible to create an **infinite loop** if your loop condition is never false!

In this level, the `while` loop has the condition `ordersGiven < 5`. Don't forget to increment `ordersGiven` inside the loop, or you will be stuck in an **infinite loop**.

So, inside your `while` loop, you need to:

1. Use `moveXY` to move down 10 meters.
2. `say` "Attack!"
3. Increment `ordersGiven` by 1.

Then, after (and outside) your `while` loop, check for an enemy, and fight if you find one.

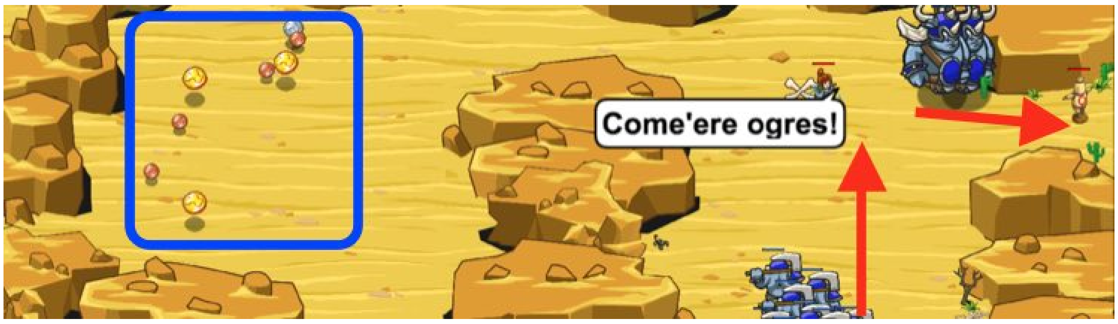
Desert Combat Solution

```
// while-loops repeat until the condition is false.

var ordersGiven = 0;
while (ordersGiven < 5) {
    // Move down 10 meters.
    hero.moveXY(hero.pos.x, hero.pos.y - 10);
    // Order your ally to "Attack!" with hero.say
    // They can only hear you if you are on the X.
    hero.say("Attack!");
    // Be sure to increment ordersGiven!
    ordersGiven += 1;
}

while(true) {
    var enemy = hero.findNearestEnemy();
    // When you're done giving orders, join the attack.
    if(enemy) {
        hero.attack(enemy);
    }
}
```

#6. Mirage Maker

Level Overview and Solutions

Intro



Use multiple `while` loops to perform different tasks.

`moveXY` to collect gold, `buildXY` a `"decoy"`, `say` to taunt the ogres, and finally `moveXY` back to the red X mark.

Default Code

```
// Lure the ogres into an ambush!

// While your gold is less than 25, collect coins.

// After the while loop, build a "decoy" at the white X.

// While your health equals maxHealth, say insults

// Then move back to the red X.
```

Overview

Use multiple `while` loops to perform a variety of tasks within the level.

First, `while` `hero.gold < 25`, collect coins.

Then, build a `"decoy"` at the bone X mark.

Then, `while` your `health` is equal to your `maxHealth`, use `say` to taunt the ogres.

After that, `moveXY` to the red X mark.

Mirage Maker Solution

```
// Lure the ogres into an ambush!

// While your gold is less than 25, collect coins.
while(hero.gold < 25) {
    var coin = hero.findNearest(hero.findItems());
    if(coin) {
        hero.moveXY(coin.pos.x, coin.pos.y);
    }
}

// After the while loop, build a "decoy" at the white X.
hero.buildXY("decoy", 71, 68);

// While your health equals maxHealth, say insults
while(hero.health == hero.maxHealth) {
    hero.say("Ogres stink!");
}

// Then move back to the red X.
hero.moveXY(22,15);
```

#7. Spinach Power

Level Overview and Solutions

Intro

Drinking spinach potions can make you stronger, but too many will make you sick.

Collect **no more than 7 potions**, then defeat the giant ogres.

Use a `while-loop` with a condition that checks how many potions you have collected.

Default Code

```
// Collect exactly 7 spinach potions.
// Then you'll be strong enough to defeat the ogres.

var potionCount = 0;

// Wrap the potion collection code inside a while loop.
// Use a condition to check the potionCount

var item = hero.findNearestItem();
if (item) {
    hero.moveXY(item.pos.x, item.pos.y);
    potionCount++;
}

// When the while loop is finished,
// Go and fight!
```

Overview

To collect `7` potions, you need to have your loop to run while `potionCount` is less than `7`.

A common error is to loop while `potionCount` is less than or equal to 7, but that will result in `8` potions being collected.

Spinach Power Solution

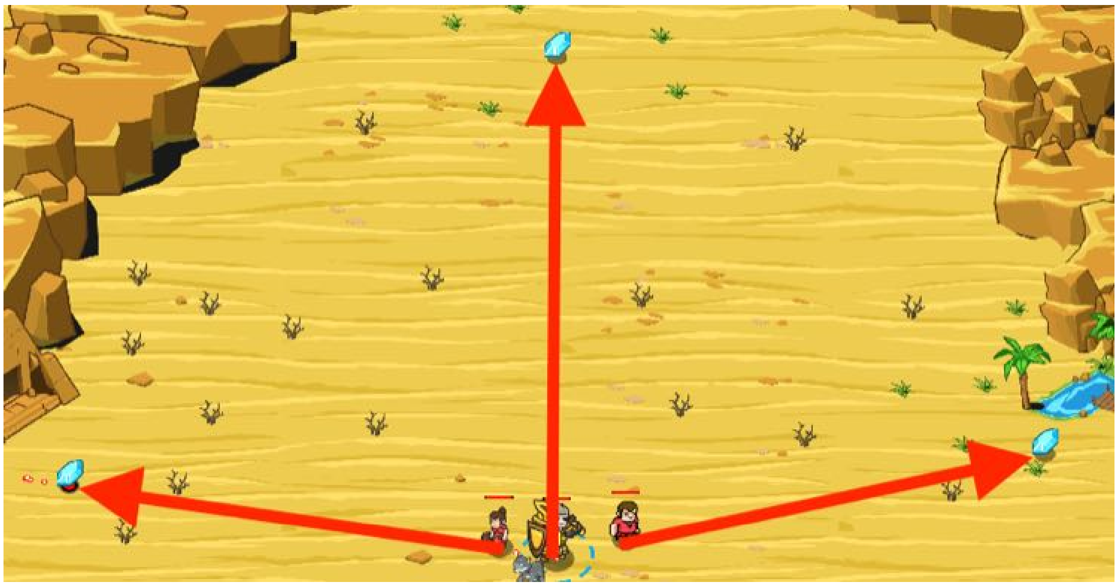
```
// Collect exactly 7 spinach potions.
// Then you'll be strong enough to defeat the ogres.

var potionCount = 0;

// Wrap the potion collection code inside a while loop.
// Use a condition to check the potionCount
while (potionCount < 7) {
    var item = hero.findNearestItem();
    if (item) {
        hero.moveXY(item.pos.x, item.pos.y);
        potionCount++;
    }
}
// When the while loop is finished,
// Go and fight!
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
    }
}
```

#8. Team Work

Level Overview and Solutions

Intro

Those gems will dissapear soon. Use peasants and the hero to collect all gems quickly.



`hero.findItems()` returns an array containing all the items your hero can see.

```
var items = hero.findItems();
var first = items[0]; // The first index is 0
var second = items[1];
var third = items[2];
```

When you assign an item to a variable, you can work with it as you did it with the `findNearestItem()` in previous levels.

Default Code

```
// Gems will disappear soon. You'll need help!

// findItems() returns an array of items.
var items = hero.findItems();

// Get the first gem from the array.
// Don't forget that the first index is 0.
var gem0 = items[0];

// # Tell Bruno to get gem0
hero.say("Bruno " + gem0);

// You can reference the gem without a variable.
hero.say("Matilda " + items[1]);

// Create a variable for the last gem, items[2]:

// Move to that gem's position using moveXY()
```

Overview

Methods like `findItems`, `findEnemies` and `findFriends` return an array filled with items, enemies, or friends).

Elements of an array are counted starting from 0, so **the first element of the array has an index of zero**. To get an element of the array use the `array[n]` syntax, where `n` is an index of the required element.

```
var enemies = hero.findEnemies();
var firstEnemy = enemies[0];
var secondEnemy = enemies[1];
```

Be careful about the length of the array. If you try to read an index that is greater than or equal to the array's length you can get an error or `undefined` value,

You can assign elements of an array to a variable.

```
var items = hero.findItems();
var firstItem = items[0];
hero.moveXY(firstItem.pos.x, firstItem.pos.y);
```

You can also use array elements without assigning them to a variable:

```
var enemies = hero.findEnemies();
hero.attack(enemies[0]);
```

Team Work Solution

```
// Gems will disappear soon. You'll need help!

// findItems() returns an array of items.
var items = hero.findItems();

// Get the first gem from the array.
// Don't forget that the first index is 0.
var gem0 = items[0];

// # Tell Bruno to get gem0
hero.say("Bruno " + gem0);

// You can reference the gem without a variable.
hero.say("Matilda " + items[1]);

// Create a variable for the last gem, items[2]:
var gem2 = items[2];

// Move to that gem's position using moveXY()
hero.moveXY(gem2.pos.x, gem2.pos.y);
```

#9. Coordinated Defense

Level Overview and Solutions

Intro

`hero.findEnemies()` returns an array of all the enemies you can see.

Attack the first enemy (index 0) in the array. Your allies will attack the others.

You need to check that the array of enemies isn't empty. One way to do this is to check that the length of the array is greater than 0:

```
var enemies = hero.findEnemies();
if (enemies.length > 0) {
    // 'enemies' contains at least one element.
}
```

Default Code

```
// Protect the peasants from the ogres.

while (true) {
    // Get an array of enemies.
    var enemies = hero.findEnemies();
    // If the array is not empty.
    if (enemies.length > 0) {
        // Attack the first enemy from "enemies" array.

        // Return to the start position.

    }
}
```

Overview

The length of an array is the number of elements contained in the array. An empty array has a length of `0`.

It is a good practice to check the length of the array if you need to read elements from it.

If you just need to read the first element, then it's enough to check that the array isn't empty.

```
var items = hero.findItems();
if (items.length) {
    // the length of "items" isn't 0.
    hero.say(items[0]);
}
```

However if you are looking for another element like the third element, then you need to check the array length more precisely:

```
if (items.length >= 3) {
    // "items" contains at least 3 elements.
    hero.say(items[2]);
}
```

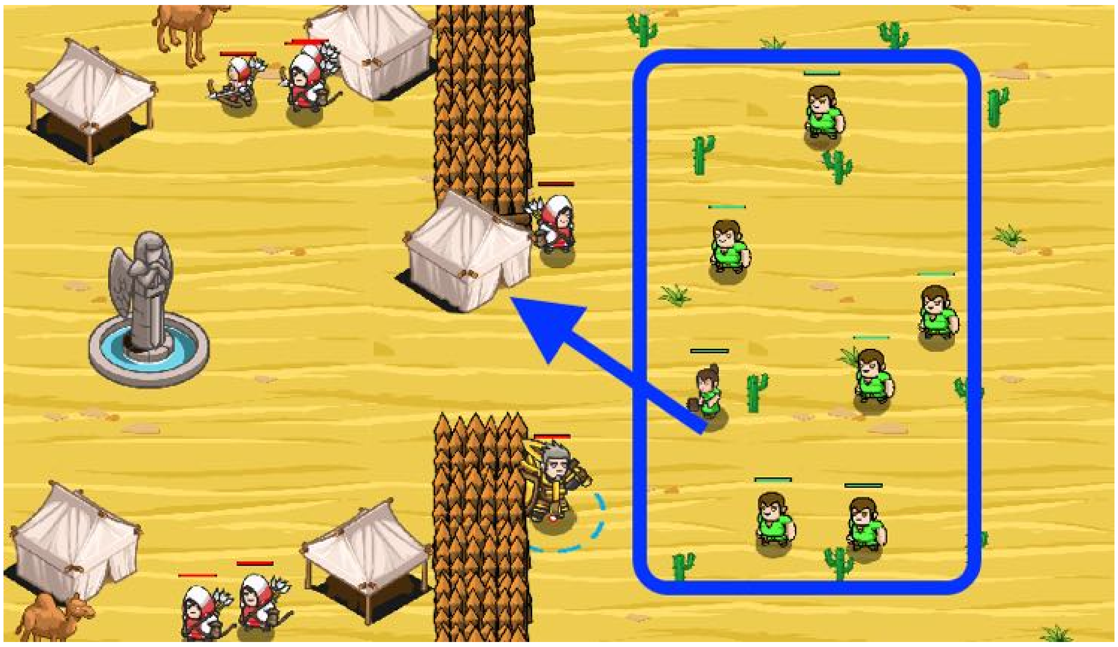
Coordinated Defense Solution

```
// Protect the peasants from the ogres.

while (true) {
    // Get an array of enemies.
    var enemies = hero.findEnemies();
    // If the array is not empty.
    if (enemies.length > 0) {
        // Attack the first enemy from "enemies" array.
        hero.attack(enemies[0]);
        // Return to the start position.
        hero.moveXY(40, 20);
    }
}
```

#10. Recruiting Queue

Level Overview and Solutions

Intro



Some peasants want to join the army.

To get the list of neutral peasants, you can use `hero.findEnemies()` .

Then call the first one from that array. To say somebody's name you can use `hero.say(unit)` or `hero.say(unit.id)` .

Because the list of peasants is changing, you should update it on each iteration of the `while-loop` . To update it, just call `findEnemies()` .

Default Code

```
// Call peasants one after another.

// Neutral units are detected as enemies.
var neutrals = hero.findEnemies();
while (true) {
    if (neutrals.length) {
        // Say the first unit in the neutrals array

    } else {
        hero.say("Nobody here");
    }
    // Reassign the neutrals variable using findEnemies()

}
```

Overview

Items can appear or be collected. Units might die or be summoned. As a result, arrays from methods like `findItems()` , `findEnemies()` and `findFriends()` can contain "old" data. To avoid the problem, you should update those arrays every loop.

```
var enemies = hero.findEnemies();
while (true) {
    hero.attack(enemies[0]);
    // "enemies" can contain out of date data.
    // Update it
    enemies = hero.findEnemies();
}
```

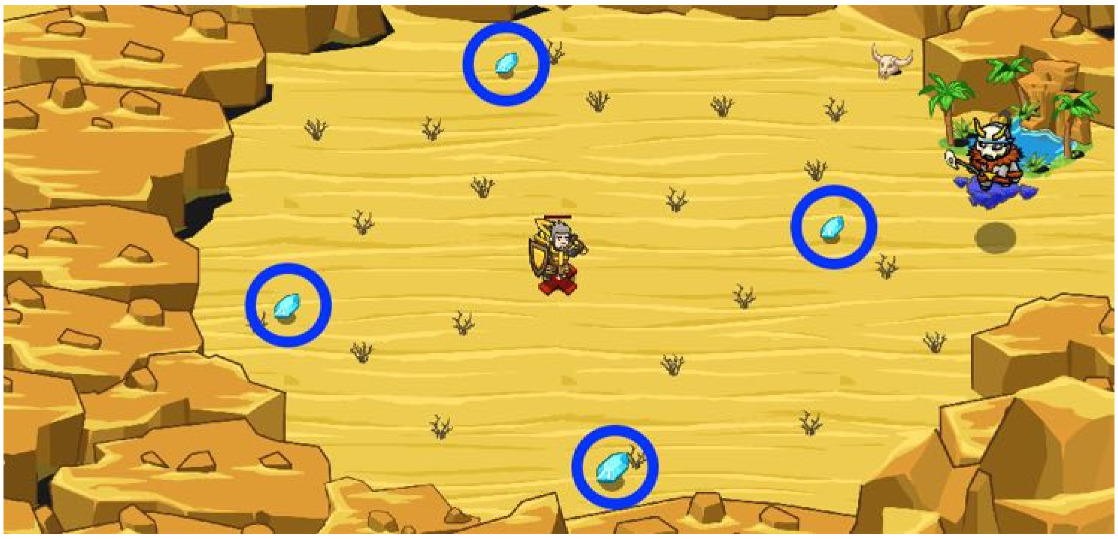
Recruiting Queue Solution

```
// Call peasants one after another.

// Neutral units are detected as enemies.
var neutrals = hero.findEnemies();
while (true) {
    if (neutrals.length) {
        // Say the first unit in the neutrals array
        hero.say(neutrals[0]);
    }
    else {
        hero.say("Nobody here");
    }
    // Reassign the neutrals variable using findEnemies()
    neutrals = hero.findEnemies();
}
```

#11. Second Gem

Level Overview and Solutions

Intro



Use `findItems()` to get an array of items and always **collect the second one from that array**.

Return to the start point to get the new bunch of items.

P.S.: Don't forget that elements are counted from 0.

Default Code

```
// One gem is safe, the others are bombs.
// But you know the answer: always take the second.

while (true) {
    var items = hero.findItems();
    // If the length of items is greater or equal to 2:

        // Move to the second item in items

    // Else:

        // Move to the center mark.

}
```

Overview

You've learned how to use `findItems()` and access elements by indexes in previous levels.

In this level, the second item in the array of items is safe to collect.

Remember that the first item is at index `0`!

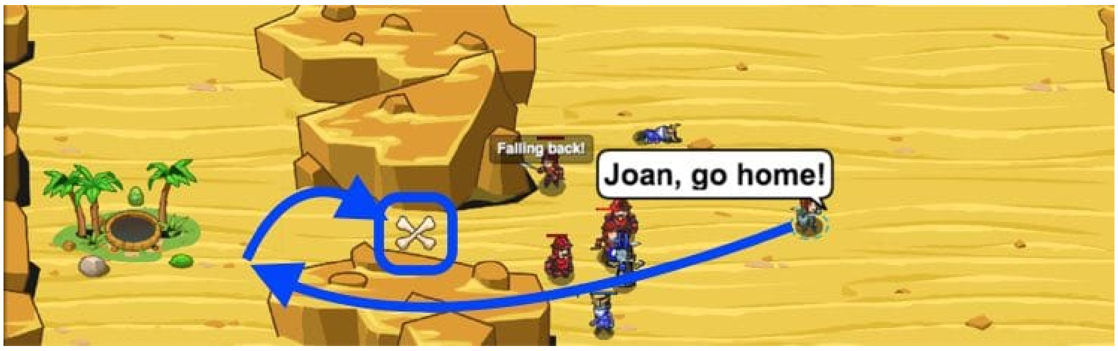
Second Gem Solution

```
// One gem is safe, the others are bombs.
// But you know the answer: always take the second.

while (true) {
    var items = hero.findItems();
    //
    if (items.length >= 2) {
        // Move to the second item in items
        hero.moveXY(items[1].pos.x, items[1].pos.y);
    }
    // Else:
    else {
        // Move to the center mark.
        hero.moveXY(40, 34);
    }
}
```

#12. Sarven Savior

Level Overview and Solutions

Intro



An `array` is a list of items.

```
// An array of strings:
var friendNames = ['Joan', 'Ronan', 'Nikita', 'Augustus'];
```

Default Code

```
// An ARRAY is a list of items.

// This array is a list of your friends' names.
var friendNames = ['Joan', 'Ronan', 'Nikita', 'Augustus'];

// Array indices start at 0, not 1!
var friendIndex = 0;

// Loop over each name in the array.
// The .length property gets the length of the array.
while (friendIndex < friendNames.length) {
    // Use square brackets to get a name from the array.
    var friendName = friendNames[friendIndex];

    // Tell your friend to go home.
    // Use + to connect two strings.
    hero.say(friendName + ', go home!');

    // Increment friendIndex to get the next name.

}

// Retreat to the oasis and build a "fence" on the X.
```

Overview

Arrays are ordered lists of data. In this level, you have an array storing the string names of your four friends.

In order to save your friends, you'll need to tell each one of them to return home in turn. You can use the provided sample code to loop over an **array**.

```
// friendNames is an array.
var friendNames = ['Joan', 'Ronan', 'Nikita', 'Augustus'];

// You can access specific elements of an array using a number surrounded by square brackets:
var name = friendNames[0];

// Causes the hero to say: "Joan"
hero.say(name)
```

You can use an index **variable** instead of a number, to access an element of the array.

To loop over all the values in an array, use a **while-loop** to increment the index variable each loop!

On each pass through the loop, you'll retrieve the friend name at that array index, then tell the friend to go home.

```
var friendNames = ['Joan', 'Ronan', 'Nikita', 'Augustus'];

// Arrays start at index 0
var friendIndex = 0;

// friendNames.length gives you the length of the friendNames array.
// The length is equal to the number of items in the array (4, in this case)

while(friendIndex < friendNames.length) {
    var friendName = friendNames[friendIndex];
    hero.say(friendName + ', go home!');
    friendIndex += 1;
}

// This while-loop will execute using friendIndex 0, then 1, 2, and 3
// Note that the length of the array is 4, but the last element is 3!
```

Sarven Savior Solution

```
// An ARRAY is a list of items.

// This array is a list of your friends' names.
var friendNames = ['Joan', 'Ronan', 'Nikita', 'Augustus'];

// Array indices start at 0, not 1!
var friendIndex = 0;

// Loop over each name in the array.
// The .length property gets the length of the array.
while (friendIndex < friendNames.length) {
    // Use square brackets to get a name from the array.
    var friendName = friendNames[friendIndex];

    // Tell your friend to go home.
    // Use + to connect two strings.
    hero.say(friendName + ', go home!');

    // Increment friendIndex to get the next name.
    friendIndex += 1;
}

// Retreat to the oasis and build a "fence" on the X.
hero.moveXY(22, 30);
hero.buildXY("fence", 30, 30);
```

#13. Bank Raid

Level Overview and Solutions

Intro

To find and iterate all enemies or items, you can use `findEnemies()` or `findItems()` and a `while-condition` loop.

```
var enemies = hero.findEnemies();
var enemyIndex = 0;
// Iterate all enemies in the enemies array.
while (enemyIndex < enemies.length) {
    enemy = enemies[enemyIndex];
    hero.attack(enemy)
    // IMPORTANT: increase enemyIndex variable.
    // Otherwise you'll get an infinite loop.
    enemyIndex++;
}
```

Default Code

```
// Wait for ogres, defeat them and collect gold.

while(true) {
    var enemies = hero.findEnemies();
    // enemyIndex is used to iterate the enemies array.
    var enemyIndex = 0;
    // While enemyIndex is less than enemies.length
    while (enemyIndex < enemies.length) {
        // Attack the enemy at enemyIndex
        var enemy = enemies[enemyIndex];
        hero.attack(enemy);
        // Increase enemyIndex by one.
        enemyIndex++;
    }
    var coins = hero.findItems();
    // coinIndex is used to iterate the coins array.
    var coinIndex = 0;
    while (coinIndex < coins.length) {
        // Get a coin from the coins array using coinIndex

        // Collect that coin.

        // Increase coinIndex by one.
        coinIndex++;
    }
}
```

Overview

You can use `while-condition` to loop (or 'iterate') over an array.

Create a variable which will be used as an index of the array.

The initial value should be zero, so first you read the element with index `0`. Then you increase that index by one and read the next element (2nd - `index` is `1`).

The `while-condition` checks if the `index` is less than the array's length.

For example:

```
var items = hero.findItems();
var itemIndex = 0;
while (itemIndex < items.length) {
    var item = items[itemIndex];
    hero.say(item);
    itemIndex += 1;
}
```

Don't forget to increment `itemIndex` each loop, or it will create an infinite loop error.

If you need to repeat those action for new groups of items (enemies, friends etc), then you can put it inside another `while-loop`. For example, you can put it inside `while-true-loop` and your hero will iterate and collect all groups of items again and again.

```
while (true) {
    var items = hero.findItems();
    var itemIndex = 0;
    while (itemIndex < items.length) {
        var item = items[itemIndex] ;
        hero.moveXY(item.pos.x, item.pos.y);
        itemIndex += 1;
    }
}
```

Bank Raid Solution

```
// Wait for ogres, defeat them and collect gold.

while(true) {
    var enemies = hero.findEnemies();
    // enemyIndex is used to iterate the enemies array.
    var enemyIndex = 0;
    // While enemyIndex is less than enemies.length
    while (enemyIndex < enemies.length) {
        // Attack the enemy at enemyIndex
        var enemy = enemies[enemyIndex];
        hero.attack(enemy);
        // Increase enemyIndex by one.
        enemyIndex++;
    }
    var coins = hero.findItems();
    // coinIndex is used to iterate the coins array.
    var coinIndex = 0;
    while (coinIndex < coins.length) {
        // Get a coin from the coins array using coinIndex
        var coin = coins[coinIndex];
        // Collect that coin.
        hero.moveXY(coin.pos.x, coin.pos.y);
        // Increase coinIndex by one.
        coinIndex++;
    }
}
```

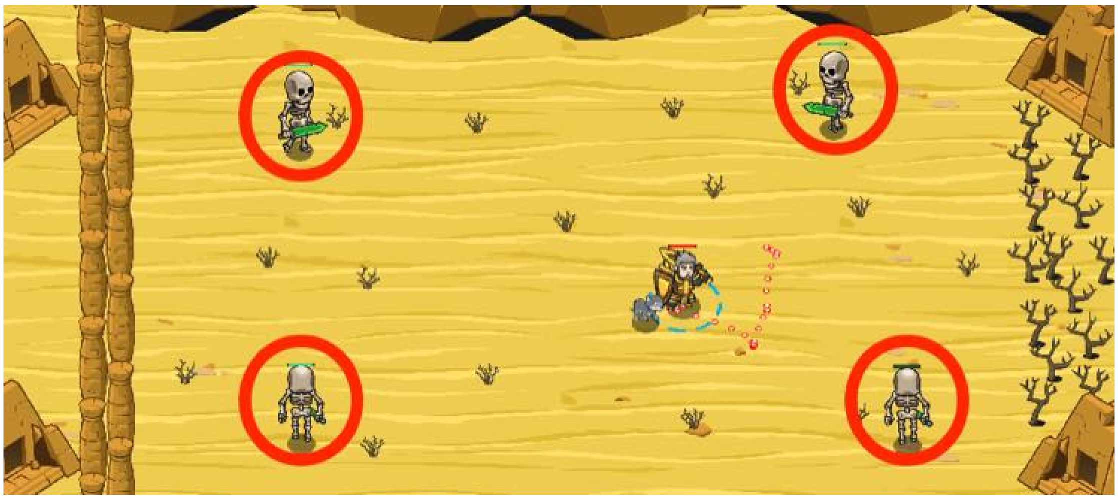
#14. Wandering Souls

Level Overview and Solutions

Intro

These wandering skeletons are not dangerous, but you should defeat them and collect the lightstones to free them from a curse.

For tough enemies, it's not enough to hit them once, so you should use `while-condition` to attack `while` their `health` is greater than `0`.



```
while (enemy.health > 0) {
    hero.attack(enemy);
}
```

You can put this loop into another `while`, and defeat all enemies one by one.

It can be used not only for enemies but for items, for example, if they don't want to be collected.

Default Code

```
// Defeat skeletons and collect lightstones.

while (true) {
    var enemies = hero.findEnemies();
    var enemyIndex = 0;
    while (enemyIndex < enemies.length) {
        var enemy = enemies[enemyIndex];
        // Attack while enemy has health.
        while (enemy.health > 0) {
            hero.attack(enemy);
        }
        enemyIndex += 1;
    }
    var items = hero.findItems();
    var itemIndex = 0;
    // Iterate over all items.
    while (itemIndex < items.length) {
        var item = items[itemIndex];
        // While the distance greater than 2:

            // Try to take the item.

        // Don't forget to increase itemIndex.
        itemIndex += 1;
    }
}
```

Overview

These wandering souls are tricky to collect. You will need to keep moving toward each orb until you're close enough to grab it!

```
while(hero.distanceTo(item) > 2) {
    // Move closer!
}
```

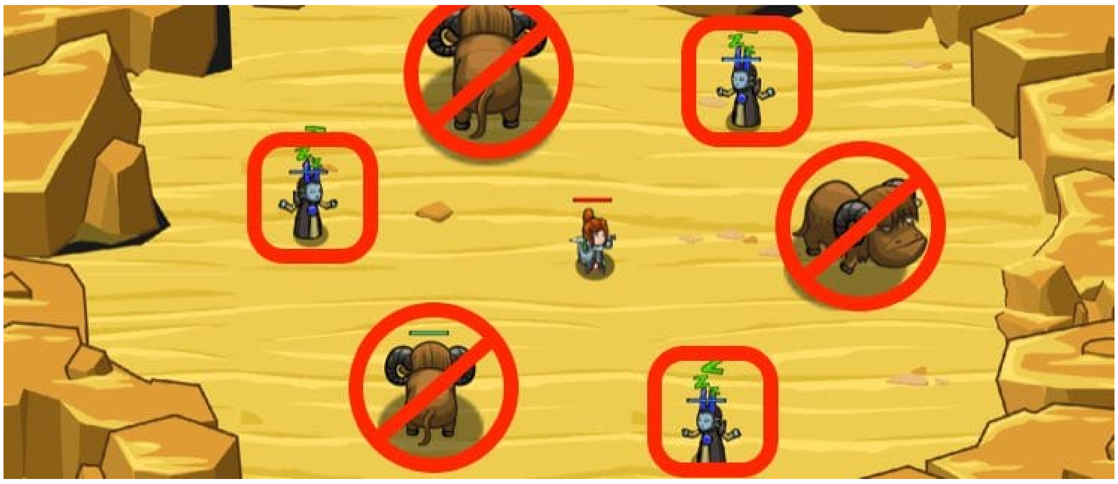
Wandering Souls Solution

```
// Defeat skeletons and collect lightstones.

while (true) {
    var enemies = hero.findEnemies();
    var enemyIndex = 0;
    while (enemyIndex < enemies.length) {
        var enemy = enemies[enemyIndex];
        // Attack while enemy has health.
        while (enemy.health > 0) {
            hero.attack(enemy);
        }
        enemyIndex += 1;
    }
    var items = hero.findItems();
    var itemIndex = 0;
    // Iterate over all items.
    while (itemIndex < items.length) {
        var item = items[itemIndex];
        // While the distance greater than 2:
        while (hero.distanceTo(item) > 2) {
            // Try to take the item.
            hero.moveXY(item.pos.x, item.pos.y);
        }
        // Don't forget to increase itemIndex.
        itemIndex += 1;
    }
}
```

#15. Lurkers

Level Overview and Solutions

Intro



`findEnemies()` gives you an array containing all enemies your hero can see:

```
var enemies = hero.findEnemies();
```

Loop through all the enemies, and attack any with type `"shaman"`.

Default Code

```
// findEnemies returns a list of all your enemies.
// Only attack shamans. Don't attack yaks!

var enemies = hero.findEnemies();
var enemyIndex = 0;

// Wrap this section in a while loop to iterate all enemies.
// While the enemyIndex is less than the length of enemies

var enemy = enemies[enemyIndex];
if (enemy.type == 'shaman') {
    while (enemy.health > 0) {
        hero.attack(enemy);
    }
}
// Remember to increment enemyIndex
```

Overview

Now that you're familiar with arrays, you can use the method `findEnemies()` to get an array containing all the enemies your hero can see.

Notice that the sample code uses another `while` loop to make sure your hero keeps attacking the shaman while it's `health` is greater than `0`!

The logic for which enemies to attack has been provided for you, you need to put it within a `while` loop, where you loop over the `enemies` to find all the `"shaman"`.

```
while(enemyIndex < enemies.length) {
    var enemy = enemies[enemyIndex];
    if(enemy.type == "shaman") {
        while(enemy.health > 0) {
            hero.attack(enemy);
        }
    }
}
```

Important: make sure you increment the `enemyIndex` every time the loop runs, even if the enemy isn't a shaman!

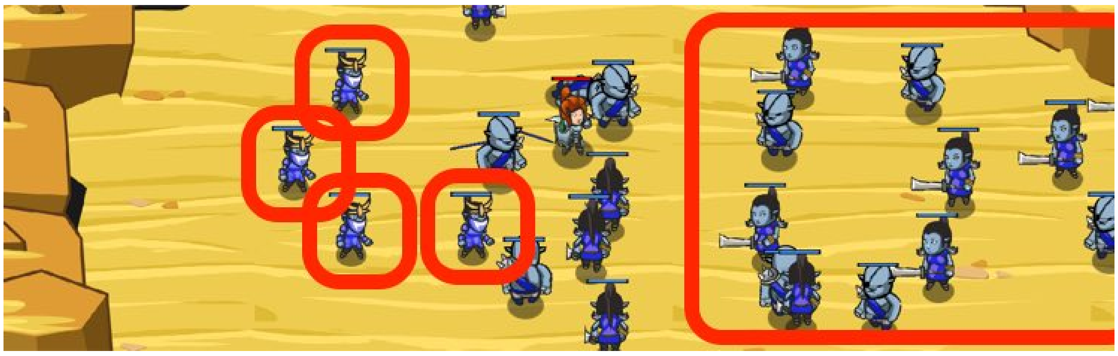
Lurkers Solution

```
// findEnemies returns a list of all your enemies.
// Only attack shamans. Don't attack yaks!

var enemies = hero.findEnemies();
var enemyIndex = 0;

// Wrap this section in a while loop to iterate all enemies.
// While the enemyIndex is less than the length of enemies
while(enemyIndex < enemies.length) {
    var enemy = enemies[enemyIndex];
    if (enemy.type == 'shaman') {
        while (enemy.health > 0) {
            hero.attack(enemy);
        }
    }
    // Remember to increment enemyIndex
    enemyIndex += 1;
}
```

#16. Preferential Treatment

Level Overview and Solutions

Intro



Throwers do a lot of damage, but have low health.

Before, you could only `findNearestEnemy`. Now, you can use `findEnemies` to find all enemies, and attack the throwers first!

First: `while` loop through all the `enemies` and attack only if `enemy.type` is `"thrower"`.

Second: `while` loop through all the `enemies` again, attacking all of them.

Default Code

```
// First, loop through all enemies...

var enemies = hero.findEnemies();
var enemyIndex = 0;
// ... but only attack "thrower" type enemies.

// Then loop through all the enemies again...
enemies = hero.findEnemies();
enemyIndex = 0;
// ... and defeat everyone who's still standing.
```

Overview

First, use a `while` loop to loop over the `enemies` array, and attack all of the enemies with type `thrower`.

Then, use another `while` loop to loop over a new `enemies` array, to attack the rest of the ogres.

Don't forget to increment the index in your loops!

Tip: when attacking, use a while loop to keep attacking if the enemy's `health` is greater than 0.

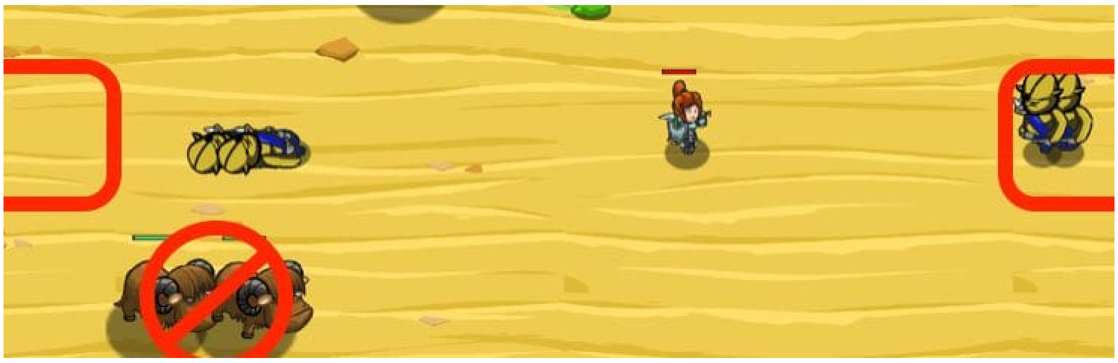
Preferential Treatment Solution

```
// First, loop through all enemies...

var enemies = hero.findEnemies();
var enemyIndex = 0;
// ... but only attack "thrower" type enemies.
while(enemyIndex < enemies.length) {
    var enemy = enemies[enemyIndex];
    if(enemy && enemy.type === "thrower") {
        hero.attack(enemy);
    }
    enemyIndex += 1;
}
// Then loop through all the enemies again...
enemies = hero.findEnemies();
enemyIndex = 0;
// ... and defeat everyone who's still standing.
while(enemyIndex < enemies.length) {
    enemy = enemies[enemyIndex];
    if(enemy) {
        hero.attack(enemy);
    }
    enemyIndex++;
}
```

#17. Sarven Shepherd

Level Overview and Solutions

Intro



Use a `while` loop to examine the array of `enemies` . Attack the enemy if it's type isn't `"sand-yak"` !

Default Code

```
// Use while loops to pick out the ogre

while(true) {
    var enemies = hero.findEnemies();
    var enemyIndex = 0;

    // Wrap this logic in a while loop to attack all enemies.
    // Find the array's length with:  enemies.length

    var enemy = enemies[enemyIndex];
    // "!=" means "not equal to."
    if (enemy.type != "sand-yak") {
        // While the enemy's health is greater than 0, attack it!

    }

    // Between waves, move back to the center.

}
```

Overview

Use `while` to loop over the array of `enemies` :

```
while(enemyIndex < enemies.length) {
    var enemy = enemies[enemyIndex];
    if(enemy.type != "sand-yak") {
        // Attack while enemy's health > 0
    }
    enemyIndex += 1;
}
```

When attacking, use a `while` loop with the condition: `enemy.health` is greater than zero to make sure you attack until the enemy is defeated.

Hint: The `moveXY()` command should be **after** (outside) your `while enemyIndex` loop, but **inside** the main **while-true** loop.

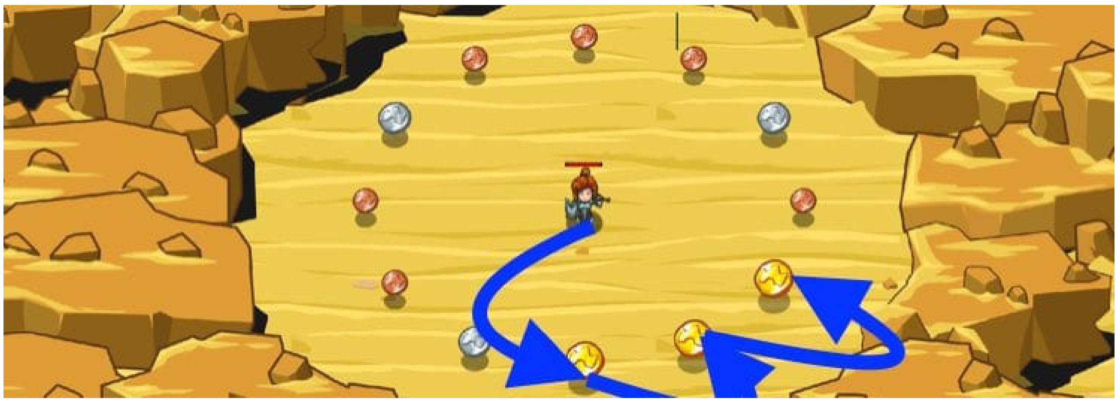
Sarven Shepherd Solution

```
// Use while loops to pick out the ogre

while(true) {
    var enemies = hero.findEnemies();
    var enemyIndex = 0;

    // Wrap this logic in a while loop to attack all enemies.
    // Find the array's length with:  enemies.length
    while(enemyIndex < enemies.length) {
        var enemy = enemies[enemyIndex];
        // "!=" means "not equal to."
        if (enemy.type != "sand-yak") {
            // While the enemy's health is greater than 0, attack it!
            while(enemy.health > 0) {
                hero.attack(enemy);
            }
        }
        enemyIndex += 1;
    }
    // Between waves, move back to the center.
    hero.moveXY(40, 32);
}
```

#18. Shine Getter

Level Overview and Solutions

Intro



Use `while` to loop over an array of coins with `findItems()`.

```
var coins = hero.findItems();
var coinIndex = 0;

while(coinIndex < coins.length) {
    var coin = coins[coinIndex];
    hero.moveXY(coin.pos.x, coin.pos.y);
    coinIndex += 1;
}
```

Default Code

```
// To grab the most gold quickly, just go after gold coins.

while(true) {
    var coins = hero.findItems();
    var coinIndex = 0;

    // Wrap this into a loop that iterates over all coins.

    var coin = coins[coinIndex];
    // Gold coins are worth 3.
    if (coin.value == 3) {
        // Only pick up gold coins.

    }
}
```

Overview

This time you'll use a `while` loop to loop over items instead of enemies, to find the gold coins.

As you can see in the sample code, the `value` property of the coin can be used to determine what kind of coin it is.

Gold coins have a `value` of `3`.

Use `moveXY` to pick up only the gold coins by moving to their `pos.x` and `pos.y`.

Shine Getter Solution

```javascript
// To grab the most gold quickly, just go after gold coins.

while(true) {
    var coins = hero.findItems();
    var coinIndex = 0;

    // Wrap this into a loop that iterates over all coins.
    while(coinIndex < coins.length) {
        var coin = coins[coinIndex];
        // Gold coins are worth 3.
        if (coin.value == 3) {
            // Only pick up gold coins.
            hero.moveXY(coin.pos.x, coin.pos.y);
        }
        coinIndex += 1;
    }
}
```

#19. Marauder

Level Overview and Solutions

Intro

These pacing mechs are full of gold. To get that gold you need to break them open!

First, use a `while` statement to collect all coins that exist.

```
var coin = hero.findNearest(hero.findItems())
while(coin) {
    # Collect the coin.
    coin = hero.findNearest(hero.findItems())
}
```

Then, attack the nearest enemy `while` its `health` is greater than `0`.

Default Code

```
// Destroy mechs and collect gold from them.

while (true) {
    var coin = hero.findNearestItem();
    // While a coin exists:

        // Move to the coin.

        // Reassign the coin variable to the nearest item.

    var enemy = hero.findNearestEnemy();
    if (enemy) {
        // While enemy's health is greater than 0.

            // Attack enemy.

    }
}
```

Overview

`while-expression` is an useful construction, which allows you do some action while something exists, for example. For example, you can watch for an enemy health while it's alive.

```
while (robot.health > 0) {
    hero.say("The robot's alive");
}
```

`while-expression` allows to check the existance of some thangs too:

```
var weakOgre = hero.findNearestEnemy();
while (weakOgre) {
    hero.attack(weakOgre);
    hero.say("One hit, one frag! Next!");
    weakOgre = hero.findNearestEnemy(); // Reassign (update) the variable.
}
```

Marauder Solution

```
// Destroy mechs and collect gold from them.

while (true) {
    var coin = hero.findNearestItem();
    // While a coin exists:
    while (coin) {
        // Move to the coin.
        hero.moveXY(coin.pos.x, coin.pos.y);
        // Reassign the coin variable to the nearest item.
        coin = hero.findNearest(hero.findItems());
    }
    var enemy = hero.findNearest(hero.findEnemies());
    if (enemy) {
        // While enemy's health is greater than 0.
        while (enemy.health > 0) {
            // Attack enemy.
            hero.attack(enemy);
        }
    }
}
```

#20. Sand Snakes

Level Overview and Solutions

Intro

`findNearest` doesn't work on this level.

Use a `while-loop` to loop over the objects in an array to find the nearest coin!

The nearest coin is the one with the smallest `distanceTo` .

Default Code

```
// This field is covered in firetraps.  Thankfully we've sent a scout ahead to find a path.  He left
    coins along the path so that if we always stick to the nearest coin, we'll avoid the traps.

// This canyon seems to interfere with your findNearest glasses!
// You'll need to find the nearest coins on your own.
while(true) {
    var coins = hero.findItems();
    var coinIndex = 0;
    var nearest = null;
    var nearestDistance = 9999;

    // Loop through all the coins to find the nearest one.
    while(coinIndex < coins.length) {
        var coin = coins[coinIndex];
        coinIndex++;
        var distance = hero.distanceTo(coin);
        // If this coin's distance is less than the nearestDistance

            // Set nearest to coin

            // Set nearestDistance to distance

    }
    // If there's a nearest coin, move to its position. You'll need moveXY so you don't cut corners and
        hit a trap.

}
```

Overview

This level teaches you how to loop over an array of items and compare them, in order to find the nearest item.

First we get an array of all the coins with `findItems` . Then we loop over the coins and keep track of two things: `nearest` and `nearestDistance` .

`nearest` is a reference to the nearest coin we have found so far. It starts off as `None` or `null` (depending on your programming language).

`nearestDistance` is the distance to the `nearest` coin. It starts out as a really high number, higher than any of the coins' distance could ever be.

As we loop through the coins, we compare the distance to the current coin to the nearestDistance so far. If it's less, we set `nearest` to be the current coin, and `nearestDistance` to be the current coin's distance.

By the end of the loop, nearest will be the coin with the smallest distance.

In future levels we will use a similar technique to find certain items or enemies based on other information, such as the lowest/highest health, or closest/farthest distance, or even best coin to collect based on it's value and distance!

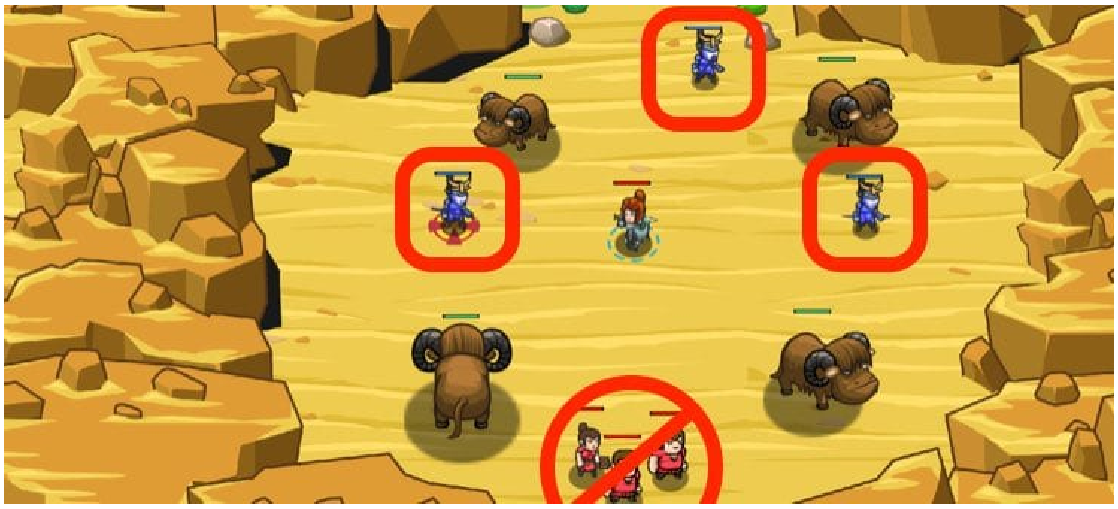
Sand Snakes Solution

```
// This field is covered in firetraps.  Thankfully we've sent a scout ahead to find a path.  He left
    coins along the path so that if we always stick to the nearest coin, we'll avoid the traps.

// This canyon seems to interfere with your findNearest glasses!
// You'll need to find the nearest coins on your own.
while(true) {
    var coins = hero.findItems();
    var coinIndex = 0;
    var nearest = null;
    var nearestDistance = 9999;

    // Loop through all the coins to find the nearest one.
    while(coinIndex < coins.length) {
        var coin = coins[coinIndex];
        coinIndex++;
        var distance = hero.distanceTo(coin);
        // If this coin's distance is less than the nearestDistance
        if(distance < nearestDistance) {
            // Set nearest to coin
            nearest = coin;
            // Set nearestDistance to distance
            nearestDistance = distance;
        }
    }
    // If there's a nearest coin, move to its position. You'll need moveXY so you don't cut corners and
        hit a trap.
    if(nearest) {
        hero.moveXY(nearest.pos.x, nearest.pos.y);
    }
}
```

#21. Odd Sandstorm

Level Overview and Solutions

Intro



Access specific elements in an array with the `[]` notation at the end of a variable name.

`everybody[enemyIndex]` returns the element in `everybody` at `enemyIndex`.

Remember, arrays are `0-indexed`, meaning the first element starts at `[0]`.

Default Code

```
// This array contains friends and ogres.
// The even elements are ogres, but the odds are friends.
var everybody = ['Yetu', 'Tabitha', 'Rasha', 'Max', 'Yazul',  'Todd'];
var enemyIndex = 0;

while (enemyIndex < everybody.length) {
    // Use square brackets to get the ogre name from the array.

    // Attack using the variable holding the ogre name.

    // Increment by 2 to skip over friends.
    enemyIndex += 2;
}

// After defeating ogres, move to the oasis.
```

Overview

The `everybody` array contains the names of both friends and enemies, in an alternating pattern.

```
// Remember to access elements of an array with square brackets:
var ogreName = everybody[enemyIndex];
```

When looping over the array, increment the index by `2` instead of 1, to skip over your friends' names.

```
// Incremeting by 2 skips every other element in an array.
enemyIndex += 2;
```

Odd Sandstorm Solution

```
// This array contains friends and ogres.
// The even elements are ogres, but the odds are friends.
var everybody = ['Yetu', 'Tabitha', 'Rasha', 'Max', 'Yazul',  'Todd'];
var enemyIndex = 0;

while (enemyIndex < everybody.length) {
    // Use square brackets to get the ogre name from the array.
    var enemy = everybody[enemyIndex];
    // Attack using the variable holding the ogre name.
    hero.attack(enemy);
    // Increment by 2 to skip over friends.
    enemyIndex += 2;
}

// After defeating ogres, move to the oasis.
hero.moveXY(36, 53);
```

#22. Mad Maxer

Level Overview and Solutions

Intro

Distracting decoys mess up your targeting.

Find the farthest enemies first, as decoys will swarm you.

The farthest enemy is the one with the most `distanceTo`.

Default Code

```
// Attack the enemy that's farthest away first.

while(true) {
    var farthest = null;
    var maxDistance = 0;
    var enemyIndex = 0;
    var enemies = hero.findEnemies();

    // Look at all the enemies to figure out which one is farthest away.
    while (enemyIndex < enemies.length) {
        var target = enemies[enemyIndex];
        enemyIndex += 1;

        // Is this enemy farther than the farthest we've seen so far?
        var distance = hero.distanceTo(target);
        if (distance > maxDistance) {
            maxDistance = distance;
            farthest = target;
        }
    }

    if (farthest) {
        // Take out the farthest enemy!
        // Keep attacking the enemy while its health is greater than 0.

    }
}
```

Overview

The goal here is to target the farthest enemies first, because those are the enemies who will attack you, while the decoys swarm in close.

The sample code shows you how to accomplish this: use a while loop to loop over all the enemies.

Initialize `maxDistance` to 0, so the first enemy in the array will be farther away than that.

Then, for each successive enemy in the array, you compare its distance to the `maxDistance` , and if it's greater, set that as the new `maxDistance` , and store that enemy in the `farthest` variable.

When you've looped through all the enemies, `farthest` will contain the enemy with the greatest distance from you.

Then, use a while loop to attack the farthest target while its health is greater than zero.

Mad Maxer Solution

```
// Attack the enemy that's farthest away first.

while(true) {
    var farthest = null;
    var maxDistance = 0;
    var enemyIndex = 0;
    var enemies = hero.findEnemies();

    // Look at all the enemies to figure out which one is farthest away.
    while (enemyIndex < enemies.length) {
        var target = enemies[enemyIndex];
        enemyIndex += 1;

        // Is this enemy farther than the farthest we've seen so far?
        var distance = hero.distanceTo(target);
        if (distance > maxDistance) {
            maxDistance = distance;
            farthest = target;
        }
    }

    if (farthest) {
        // Take out the farthest enemy!
        // Keep attacking the enemy while its health is greater than 0.
        while(farthest.health > 0) {
            hero.attack(farthest);
        }
    }
}
```

#23. Brittle Morale

Level Overview and Solutions

Intro

You only have one arrow, so you need to defeat the ogre leader first!

Loop over all the `enemies` and compare their `health`. The one with the most `health` is the boss!

Default Code

```
// You have one arrow. Make it count!

// This should return the enemy with the most health.
function findStrongestEnemy(enemies) {
    var strongest = null;
    var strongestHealth = 0;
    var enemyIndex = 0;
    // While enemyIndex is less than the length of enemies:

        // Set an enemy variable to enemies[enemyIndex]

        // If enemy.health is greater than strongestHealth

            // Set strongest to enemy
            // Set strongestHealth to enemy.health

        // Increment enemyIndex


    return strongest;
}

var enemies = hero.findEnemies();
var leader = findStrongestEnemy(enemies);
if (leader) {
    hero.say(leader);
}
```

Overview

To find the leader ogre, you need to implement a function to find the ogre with the most health.

The function should take in an array containing enemies and should return a single ogre.

Use the `enemy.health` value as the point of comparison.

Remember how to iterate over an array:

```
var array = ["A", "B", "C", "D", "E"];
var index = 0;
while(index < array.length) {
    hero.say(array[index]); // First the hero says "A", then "B", and so on.
    // Increment the index.
    // This lets you check every index, and adds an end condition for the while loop.
    index += 1;
}
hero.say("That's the ABCs!");
```

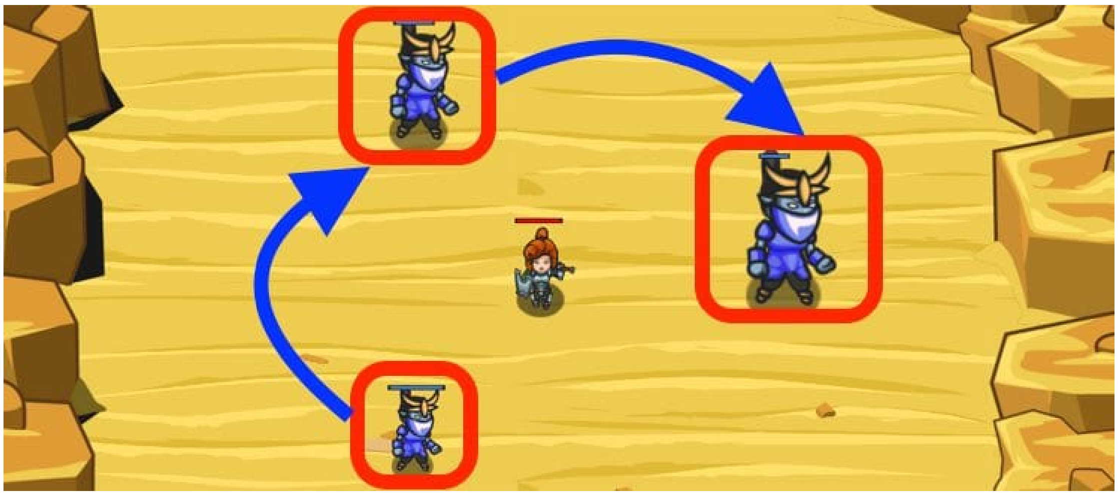
Brittle Morale Solution

```
// You have one arrow. Make it count!

// This should return the enemy with the most health.
function findStrongestEnemy(enemies) {
    var strongest = null;
    var strongestHealth = 0;
    var enemyIndex = 0;

    // While enemyIndex is less than the length of enemies:
    while (enemyIndex < enemies.length) {
        // Set an enemy variable to enemies[enemyIndex]
        var enemy = enemies[enemyIndex];
        // If enemy.health is greater than strongestHealth
        if (enemy.health > strongestHealth) {
            // Set strongest to enemy
            // Set strongestHealth to enemy.health
            strongest = enemy;
            strongestHealth = enemy.health;
        }
        // Increment enemyIndex
        enemyIndex += 1;
    }
    return strongest;
}

var enemies = hero.findEnemies();
var leader = findStrongestEnemy(enemies);
if (leader) {
    hero.say(leader);
}
```

#24. Mad Maxer Strikes Back

Level Overview and Solutions

Intro



The smallest ogres are the most dangerous.

Be sure to attack the ogres with the least health, first.

Default Code

```
// The smaller ogres here do more damage!
// Attack the ogres with the least health first.
while(true) {
    var weakest = null;
    var leastHealth = 99999;
    var enemyIndex = 0;
    var enemies = hero.findEnemies();

    // Loop through all enemies.

        // If an enemy's health is less than leastHealth,

            // make it the weakest and set leastHealth to its health.

    if (weakest) {
        // Attack the weakest ogre.

    }
}
```

Overview

In this level, you should attack the enemy with the least health first.

The sample code starts out by initializing weakest to `None` or `null` (depends on your language) and `leastHealth` to a value that is higher than any enemy's health could possibly be.

You should use a `while` loop to examine each enemy, comparing its health to `leastHealth`. If an enemy's health is lower than `leastHealth`, you set `weakest` to be that enemy, and update `leastHealth` to be that enemy's health.

This way, at the end of your loop, `weakest` will be the enemy with the lowest health.

Mad Maxer Strikes Back Solution

```javascript
// The smaller ogres here do more damage!
// Attack the ogres with the least health first.
while(true) {
    var weakest = null;
    var leastHealth = 99999;
    var enemyIndex = 0;
    var enemies = hero.findEnemies();

    // Loop through all enemies.
    while(enemyIndex < enemies.length) {
        var enemy = enemies[enemyIndex];
        enemyIndex += 1;
        // If an enemy's health is less than leastHealth,
        if(enemy.health < leastHealth) {
            // make it the weakest and set leastHealth to its health.
            weakest = enemy;
            leastHealth = enemy.health;
        }
    }

    if (weakest) {
        // Attack the weakest ogre.
        hero.attack(weakest);
    }
}
```

#25. Wishing Well

Level Overview and Solutions

Intro

You need exactly 104 gold.

Use the `sumCoinValues()` function to get the total value of coins.

If there is not enough gold say `"Non satis"`.

If there is too much gold say `"Nimis"`.

If there is exactly `104` gold, then collect the coins.

Hint: check out the code for `sumCoinValues()` to understand how it works!

Default Code

```
// You need exactly 104 gold.

var less = "Nimis";
var more = "Non satis";
var requiredGold = 104;

// This function calculates the sum of all coin values.
function sumCoinValues(coins) {
    var coinIndex = 0;
    var totalValue = 0;
    // Iterate all coins.
    while (coinIndex < coins.length) {
        totalValue += coins[coinIndex].value;
        coinIndex++;
    }
    return totalValue;
}

function collectAllCoins() {
    var item = hero.findNearest(hero.findItems());
    while (item) {
        hero.moveXY(item.pos.x, item.pos.y);
        item = hero.findNearest(hero.findItems());
    }
}

while (true) {
    var items = hero.findItems();
    // Get the total value of coins.
    var goldAmount = sumCoinValues(items);
    // If there are coins, then goldAmount isn't zero.
    if (goldAmount !== 0) {
        // If goldAmount is less than requiredGold
        // Then say "Non satis".

        // If goldAmount is greater than requiredGold
        // Then say "Nimis".

        // If goldAmount is exactly equal to requiredGold
        // If there is exactly 104 gold, then collect all coins.

    }
}
```

Overview

The simplest example of a **summary function** is the calculation of the sum of an array of numbers:

```
function sumNumbers(array) {
    var index = 0;
    var total = 0;
    while (index < array.length) {
        total += array[index];
        index++;
    }
}
```

We iterate all elements of the array and add them to the variable `total`.

In this level, we have an array of objects (coins), so we need to calculate the total of each coin's `value` property.

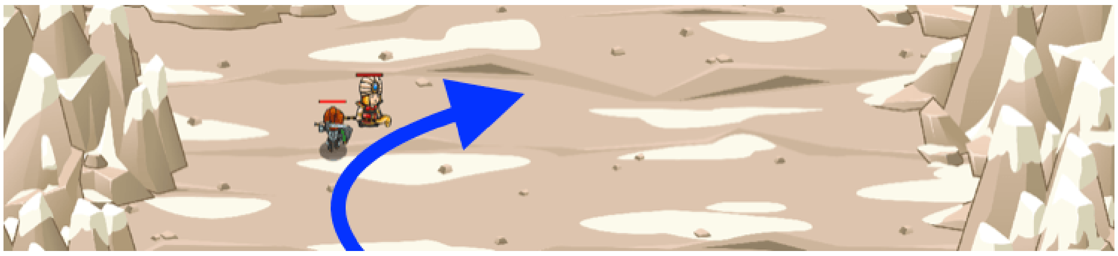
Wishing Well Solution

```
// You need exactly 104 gold.

var less = "Nimis";
var more = "Non satis";
var requiredGold = 104;

// This function calculates the sum of all coin values.
function sumCoinValues(coins) {
    var coinIndex = 0;
    var totalValue = 0;
    // Iterate all coins.
    while (coinIndex < coins.length) {
        totalValue += coins[coinIndex].value;
        coinIndex++;
    }
    return totalValue;
}

function collectAllCoins() {
    var item = hero.findNearest(hero.findItems());
    while (item) {
        hero.moveXY(item.pos.x, item.pos.y);
        item = hero.findNearest(hero.findItems());
    }
}

while (true) {
    var items = hero.findItems();
    // Get the total value of coins.
    var goldAmount = sumCoinValues(items);
    // If there are coins, then goldAmount isn't zero.
    if (goldAmount !== 0) {
        // If goldAmount is less than requiredGold
        // Then say "Non satis".
        if (goldAmount < requiredGold) {
            hero.say(more);
        }
        // If goldAmount is greater than requiredGold
        // Then say "Nimis".
        if (goldAmount > requiredGold) {
            hero.say(less);
        }
        // If goldAmount is exactly equal to requiredGold
        // If there is exactly 104 gold, then collect all coins.
        if (goldAmount == requiredGold) {
            collectAllCoins();
        }
    }
}
```

#26. Crag Tag

Level Overview and Solutions

Intro



`move` is different from `moveXY`!

`move(pos)` has a single argument instead of `moveXY(x, y)`'s two arguments.

You can use it to move to a coin's position like: `move(coin.pos)` instead of `moveXY(coin.pos.x, coin.pos.y)`

Finally, `moveXY` moves until the destination is reached. `move` only moves a single step towards the target destination. Now you can do things inbetween movement commands!

Default Code

```
// Catch up to Pender Spellbane to learn her secrets.

while(true) {
    // Pender is the only friend here, so she's always the nearest.
    var pender = hero.findNearest(hero.findFriends());

    if (pender) {
        // moveXY() will move to where Pender is,
        // but she'll have moved away by the time you get there.
        hero.moveXY(pender.pos.x, pender.pos.y);

        // move() only moves one step at a time,
        // so you can use it to track your target.
        //hero.move(pender.pos);
    }
}
```

Overview

moveXY vs move

This level shows the difference between the old `moveXY` movement and the new `move` movement.

(x,y) vs (pos)

With `move`, you specify a **position** to move to.

Positions are objects with an `x` property and a `y` property. You've used these before, with `moveXY` like:

```
hero.moveXY(coin.pos.x, coin.pos.y)
```

but with `move` you'd just pass the `pos` object as the argument, like:

```
hero.move(coin.pos)
```

Block execution or continue execution?

With `moveXY` **your program will stop executing** until your hero has reached the specified `(x,y)` location.

With `move` your hero will **move toward** the `pos` you pass in, but **your program will continue to execute**.

This means that your hero will take a few steps in the direction of `pos` , but then your program will continue to run, so you can interrupt that movement by taking different actions in the next loops of your code.

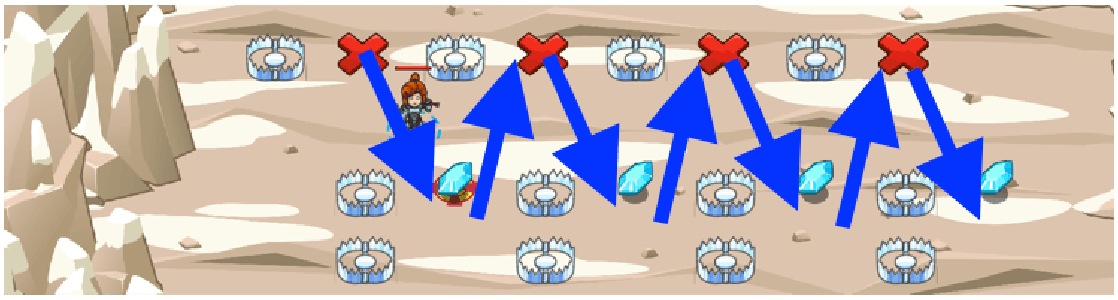
Crag Tag Solution

```
// Catch up to Pender Spellbane to learn her secrets.

while(true) {
    // Pender is the only friend here, so she's always the nearest.
    var pender = hero.findNearest(hero.findFriends());

    if (pender) {
        // moveXY() will move to where Pender is,
        // but she'll have moved away by the time you get there.
        //hero.moveXY(pender.pos.x, pender.pos.y);

        // move() only moves one step at a time,
        // so you can use it to track your target.
        hero.move(pender.pos);
    }
}
```

#27. Slalom

Level Overview and Solutions

Intro



When using `move` you need to construct an object literal to pass as an argument.

You can specify an object literal like this: `pos = { "x": 20, "y": 35 }` .

An object literal is made up of the keys associated with it. For example `pos.x` would return `20` while `pos.y` would return `35` .

Default Code

```
// Use object literals to walk the safe path and collect the gems.
// You cannot use moveXY() on this level! Use move() to get around.
var gems = hero.findItems();

while (hero.pos.x < 20) {
    // move() takes objects with x and y properties, not just numbers.
    hero.move({'x': 20, 'y': 35});
}

while (hero.pos.x < 25) {
    // A gem's position is an object with x and y properties.
    var gem0 = gems[0];
    hero.move(gem0.pos);
}

// While your x is less than 30,
// Use an object to move to 30, 35.

// While your x is less than 35,
// Move to the position of gems[1].

// Get to the last couple of gems yourself!
```

Overview

When using `move` , you may need to construct an **object literal** to pass as its argument.

```
// this is an object literal
var ob = { "x": 20, "y": 35 };
```

Remember that `move` does not block execution of your code? In this level you will use `while` loops to keep moving toward a position while your `pos.x` is less than a certain number.

To move to an X mark, pass an **object literal** to the `move` method.

To move to a gem, use the gem's `pos` object as the argument for `move` .

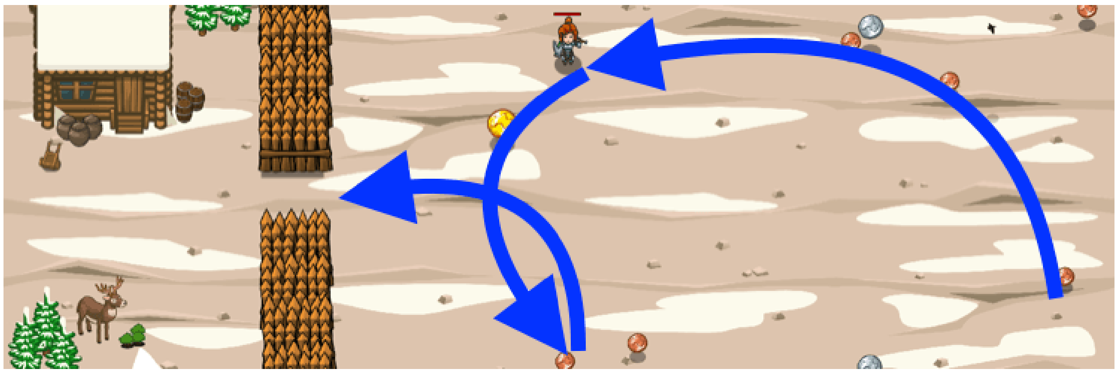
Slalom Solution

```
// Use object literals to walk the safe path and collect the gems.
// You cannot use moveXY() on this level! Use move() to get around.
var gems = hero.findItems();

while (hero.pos.x < 20) {
    // move() takes objects with x and y properties, not just numbers.
    hero.move({'x': 20, 'y': 35});
}

while (hero.pos.x < 25) {
    // A gem's position is an object with x and y properties.
    var gem0 = gems[0];
    hero.move(gem0.pos);
}

// While your x is less than 30,
// Use an object to move to 30, 35.
while(hero.pos.x < 30) {
    hero.move({'x': 30, 'y': 35});
}

// While your x is less than 35,
// Move to the position of gems[1].
while(hero.pos.x < 35) {
    var gem1 = gems[1];
    hero.move(gem1.pos);
}


// Get to the last couple of gems yourself!
while(hero.pos.x < 40) {
    hero.move({'x': 40, 'y':35});
}

while(hero.pos.x < 45) {
    var gem2 = gems[2];
    hero.move(gem2.pos);
}

while(hero.pos.x < 50) {
    hero.move({'x': 50, 'y':35});
}

while(hero.pos.x < 55) {
    var gem3 = gems[3];
    hero.move(gem3.pos);
}
```

#28. Ogre Gorge Gouger

Level Overview and Solutions

Intro



Use `findItems()` for an array of all the coins. `findNearest(array)` to find the nearest element in array.

Remember how to construct object literals: `{"x": 16, "y": 38}`.

Default Code

```
// You only have 20 seconds until the ogre horde arrives!
// Grab as much gold as you can, then retreat to your base and wall it off!
while(hero.time < 20) {
    // Collect coins
    hero.say("I should pick up a coin");

}

while(hero.pos.x > 16) {
    // Retreat behind the fence
    hero.say("I should retreat");

}

// Build a fence to keep the ogres out.
```

Overview

Use `findItems()` to find coins with advanced glasses. See if you can optimize your path to collect more than 60 coins before the ogres show up!

Use object literals like `{"x": 16, "y": 38 }` to move to a specific spot, such as behind the fences.

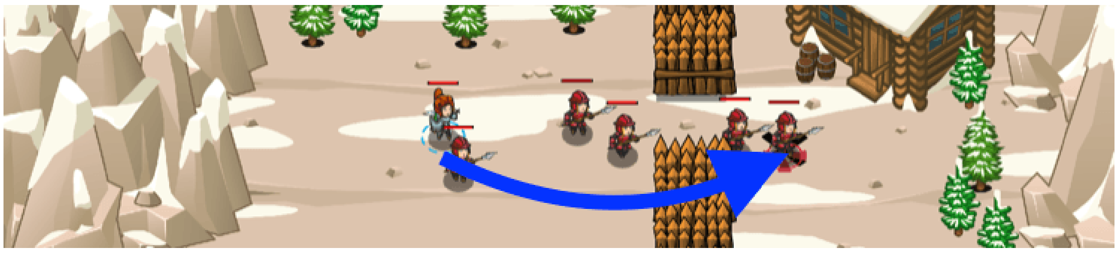
Ogre Gorge Gouger Solution

```
// You only have 20 seconds until the ogre horde arrives!
// Grab as much gold as you can, then retreat to your base and wall it off!
while(hero.time < 20) {
    // Collect coins
    var coin = hero.findNearest(hero.findItems());
    hero.move(coin.pos);
}

while(hero.pos.x > 16) {
    // Retreat behind the fence
    hero.move({"x": 15, "y": 38});
}

// Build a fence to keep the ogres out.
hero.buildXY("fence", 20, 37);
```

#29. Cloudrip Commander

Level Overview and Solutions

Intro



In this level you have access to the Boss Star I. You can now `summon` `"soldier"` and `"archer"` allies.

You can now also `command` your `soldier` and `archer` allies.

Default Code

```
// Summon some soldiers, then direct them to your base.

// Each soldier costs 20 gold.
while (hero.gold > hero.costOf("soldier")) {
    hero.summon("soldier");
}

var soldiers = hero.findFriends();
var soldierIndex = 0;
// Add a while loop to command all the soldiers.

var soldier = soldiers[soldierIndex];
hero.command(soldier, "move", {x: 50, y: 40});

// Go join your comrades!
```

Overview

This level introduces the **Boss Star**, an item that allows you to summon and command allies.

The **Boss Star I** lets you summon `"soldier"` allies and command `"soldier"` and `"archer"` allies.

The sample code first demonstrates how to use the `costOf` and `summon` methods to summon `"soldier"` allies if you have enough gold to do so.

Then, it uses `findFriends` to get an array of your allies, and shows how to command an ally to `"move"` with the `command` method.

You need to take this `command` code and put it inside a `while` loop, so you give commands to all the soldiers, instead of just one.

Then, use another `while` loop with `move` to move your hero to the X mark.

Cloudrip Commander Solution

```
// Summon some soldiers, then direct them to your base.

// Each soldier costs 20 gold.
while (hero.gold > hero.costOf("soldier")) {
    hero.summon("soldier");
}

var soldiers = hero.findFriends();
var soldierIndex = 0;
// Add a while loop to command all the soldiers.
while(soldierIndex < soldiers.length) {
    var soldier = soldiers[soldierIndex];
    hero.command(soldier, "move", {x: 50, y: 40});
    soldierIndex += 1;
}

// Go join your comrades!
var target = {"x": 48, "y": 40};
while(hero.distanceTo(target)) {
    hero.move(target);
}
```

#30. Mountain Mercenaries

Level Overview and Solutions

Intro

Use `summon` and `command` to protect yourself while gathering coins.

Default Code

```
// Gather coins to summon soldiers and have them attack the enemy.

while(true) {
    // Move to the nearest coin.
    // Use move instead of moveXY so you can command constantly.

    hero.say("I need coins!");

    // If you have funds for a soldier, summon one.
    if (hero.gold > hero.costOf("soldier")) {
        hero.say("I should summon something here!");
    }
    var enemy = hero.findNearest(hero.findEnemies());
    if (enemy) {
        // Loop over all your soldiers and order them to attack.

        var soldiers = hero.findFriends();
        var soldierIndex = 0;
        var soldier = soldiers[soldierIndex];

        // Use the 'attack' command to make your soldiers attack.
        //hero.command(soldier, "attack", enemy);
    }
}
```

Overview

It's time to practice summoning and commanding soldiers!

The sample code shows you how to use `costOf` to check if you have enough gold to summon a soldier.

If you do, use `summon` to summon a `"soldier"`.

Then, use a `while` loop to loop through the array of `soldiers` and use `command` with `"attack"` to have your soldiers attack any enemies.

The sample code's comments show you how to do this.

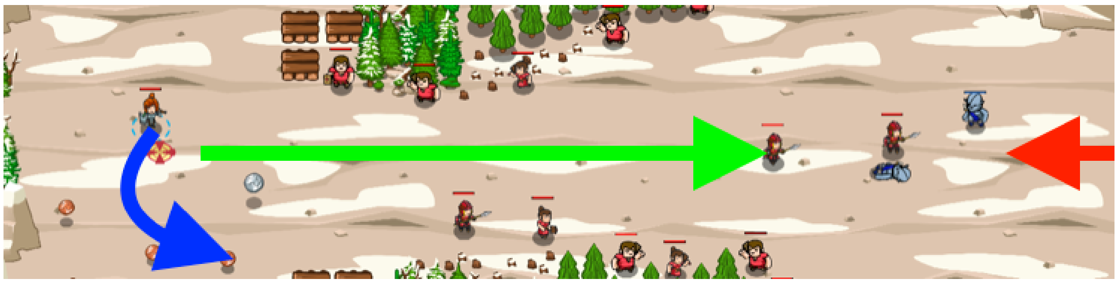
Mountain Mercenaries Solution

```
// Gather coins to summon soldiers and have them attack the enemy.

while(true) {
    // Move to the nearest coin.
    // Use move instead of moveXY so you can command constantly.
    var coin = hero.findNearest(hero.findItems());
    if(coin) {
        hero.move(coin.pos);
    }

    // If you have funds for a soldier, summon one.
    if (hero.gold > hero.costOf("soldier")) {
        hero.summon("soldier");
    }

    var enemy = hero.findNearest(hero.findEnemies());
    if (enemy) {
        // Loop over all your soldiers and order them to attack.

        var soldiers = hero.findFriends();
        var soldierIndex = 0;
        while(soldierIndex < soldiers.length) {
            var soldier = soldiers[soldierIndex];
            soldierIndex += 1;

            // Use the 'attack' command to make your soldiers attack.
            hero.command(soldier, "attack", enemy);
        }
    }
}
```

#31. Timber Guard

Level Overview and Solutions

Intro



A `for-loop` is similar to a `while-loop`. There is more setup, but it can be used to accomplish the same thing.

Default Code

```
while(true) {
    // Collect gold.

    // If you have enough gold, summon a soldier.

    // Use a for-loop to command each soldier.
    var friends = hero.findFriends();
    // For-loops have 3 parts, separated by semicolons.
    // for(initialization; condition; expression)
    // Initialization is done at the start of the first loop.
    // The loops continue while condition is true.
    for(var friendIndex = 0; friendIndex < friends.length; friendIndex++) {
        var friend = friends[friendIndex];
        if(friend.type == "soldier") {
            var enemy = friend.findNearestEnemy();
            // If there's an enemy, command her to attack.
            // Otherwise, move her to the right side of the map.

        }
    }
}
```

Overview

This level introduces `for` loops. They can be quite different depending on the language you're using, so look at the comments in the level's default code for help.

Remember that you command your soldiers with `command(soldier, "attack", enemy)` or `command(soldier, "move", pos)`.

Your hero should stay behind and collect coins. If you try to fight the ogres personally, you may attract the attention of ogre assassins!

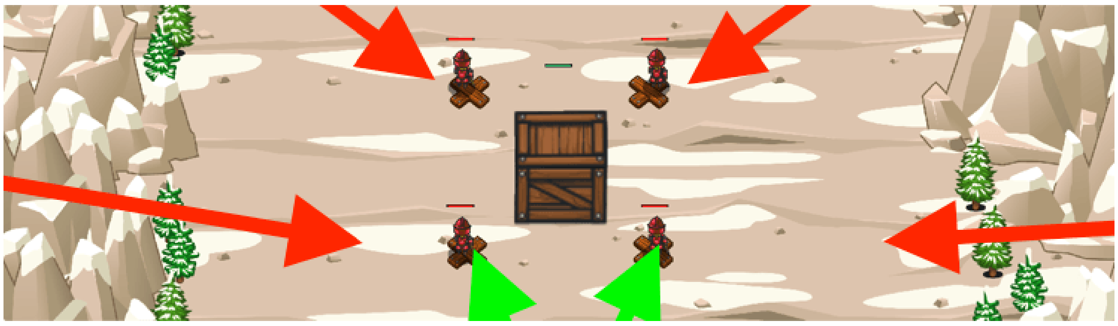
Timber Guard Solution

```
while(true) {
    // Collect gold.
    var coin = hero.findNearest(hero.findItems());
    if(coin) {
        hero.move(coin.pos);
    }

    // If you have enough gold, summon a soldier.
    if(hero.gold >= hero.costOf("soldier")) {
        hero.summon("soldier");
    }

    // Use a for-loop to command each soldier.
    var friends = hero.findFriends();
    // For-loops have 3 parts, separated by semicolons.
    // for(initialization; condition; expression)
    // Initialization is done at the start of the first loop.
    // The loops continue while condition is true.
    for(var friendIndex = 0; friendIndex < friends.length; friendIndex++) {
        var friend = friends[friendIndex];
        if(friend.type == "soldier") {
            var enemy = friend.findNearestEnemy();
            // If there's an enemy, command her to attack.
            // Otherwise, move her to the right side of the map.
            if(enemy) {
                hero.command(friend, "attack", enemy);
            } else {
                var rightPos = {"x": 83, "y": 45};
                hero.command(friend, "move", rightPos);
            }
        }
    }
}
```

#32. Zoo Keeper

Level Overview and Solutions

Intro



Do not let the ogres break open the box!

Default Code

```
// Protect the cage.
// Put a soldier at each X.
var points = [];
points[0] = {x: 33, y: 42};
points[1] = {x: 47, y: 42};
points[2] = {x: 33, y: 26};
points[3] = {x: 47, y: 26};

// 1. Collect 80 gold.

// 2. Build 4 soldiers.
for(var i=0; i < 4; i++) {
    hero.summon("soldier");
}

// 3. Send your soldiers into position.
while(true) {
    var friends = hero.findFriends();
    for(var j=0; j < friends.length; j++) {
        var point = points[j];
        var friend = friends[j];
        var enemy = friend.findNearestEnemy();
        if(enemy && enemy.team == "ogres" && friend.distanceTo(enemy) < 5) {
            // Command friend to attack.

        } else {
            // Command friend to move to point.

        }
    }
}
```

Overview

This level shows how you can use a range of numbers with for loops, to access multiple related arrays.

You also see how to have a soldier defend a certain location.

When collecting coins, you can stop at 80, because each soldier costs 20.

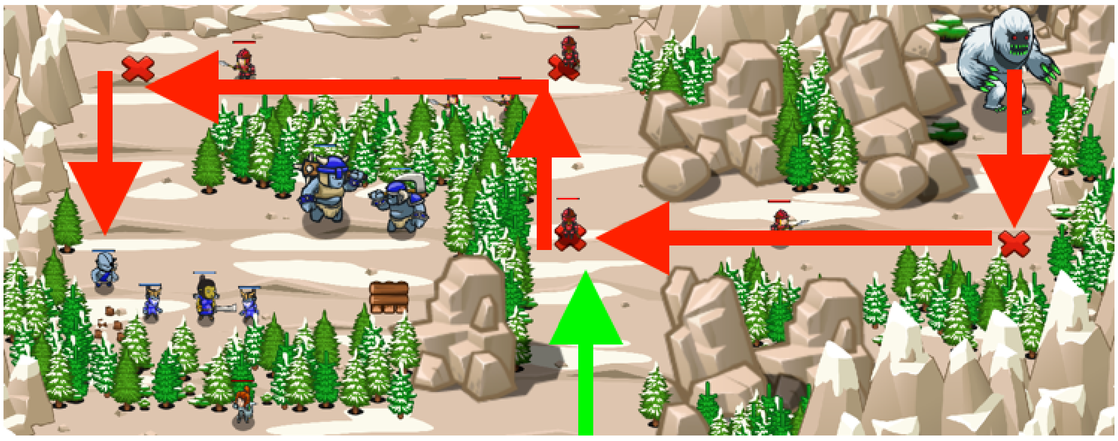
Zoo Keeper Solution

```
// Protect the cage.
// Put a soldier at each X.
var points = [];
points[0] = {x: 33, y: 42};
points[1] = {x: 47, y: 42};
points[2] = {x: 33, y: 26};
points[3] = {x: 47, y: 26};

// 1. Collect 80 gold.
while(hero.gold < 80) {
    var coin = hero.findNearest(hero.findItems());
    if(coin) {
        hero.move(coin.pos);
    }
}

// 2. Build 4 soldiers.
for(var i=0; i < 4; i++) {
    hero.summon("soldier");
}

// 3. Send your soldiers into position.
while(true) {
    var friends = hero.findFriends();
    for(var j=0; j < friends.length; j++) {
        var point = points[j];
        var friend = friends[j];
        var enemy = friend.findNearestEnemy();
        if(enemy && enemy.team == "ogres" && friend.distanceTo(enemy) < 5) {
            // Command friend to attack.
            hero.command(friend, "attack", enemy);
        } else {
            // Command friend to move to point.
            hero.command(friend, "move", point);
        }
    }
}
```

#33. Noble Sacrifice

Level Overview and Solutions

Intro



In this level you want to use `for-loops` again.

Remember your soldiers fondly.

Default Code

```
// Collect 80 gold

// Build 4 soldiers to use as bait

// Send your soldiers into position
var points = [];
points[0] = { x: 13, y: 73 };
points[1] = { x: 51, y: 73 };
points[2] = { x: 51, y: 53 };
points[3] = { x: 90, y: 52 };
var friends = hero.findFriends();

// Use a for-loop to loop over i from 0 to 4
// Match the friends to the points and command them to move
```

Overview

For this level you'll need a `for` loop.

```
for(var i=0; i < 4; i++) {
    var friend = friends[i];
    var point = points[i];

}
```

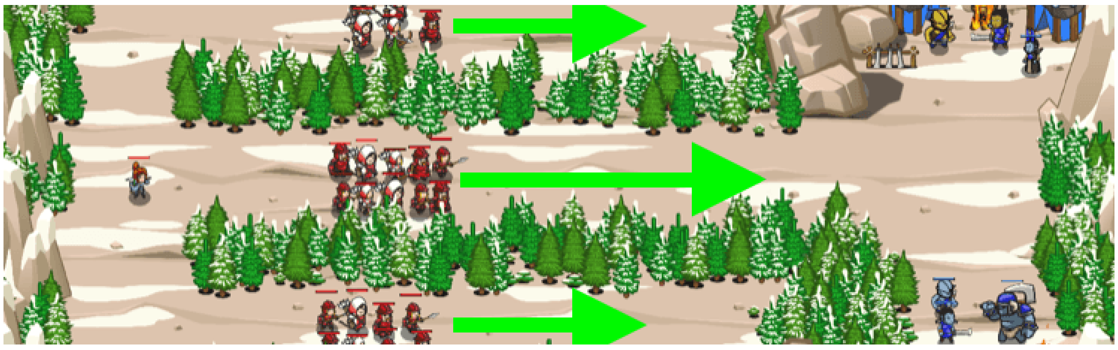
Noble Sacrifice Solution

```
// Collect 80 gold
while(hero.gold < 80) {
    var coin = hero.findNearest(hero.findItems());
    if(coin) {
        hero.move(coin.pos);
    }
}

// Build 4 soldiers to use as bait
for(var i=0; i < 4; i++) {
    hero.summon("soldier");
}

// Send your soldiers into position
var points = [];
points[0] = { x: 13, y: 73 };
points[1] = { x: 51, y: 73 };
points[2] = { x: 51, y: 53 };
points[3] = { x: 90, y: 52 };
var friends = hero.findFriends();

// Use a for-loop to loop over i from 0 to 4
// Match the friends to the points and command them to move
for(var j=0; j < friends.length; j++) {
    var point = points[j];
    var friend = friends[j];
    hero.command(friend, "move", point);
}
```

#34. Hunting Party

Level Overview and Solutions

Intro



Use `findFriends()` and a `for-loop`.

Command your troops to advance or attack the ogres!

Default Code

```
// Command your troops to move east and attack any ogres they see.
// Use for-loops and findFriends.
// You can use findNearestEnemy() on your soldiers to get their nearest enemy instead of yours.
```

Overview

Use `findFriends()` to get an array of your friends. Then use a `for` loop to give them all commands.

Command them to move to the right (add to the x position) or fight if they see ogres.

Hint: Adding smaller values to the units' x position can limit their speed. This might help your faster units stay in formation with your slower units, which can affect the outcome of a fight.

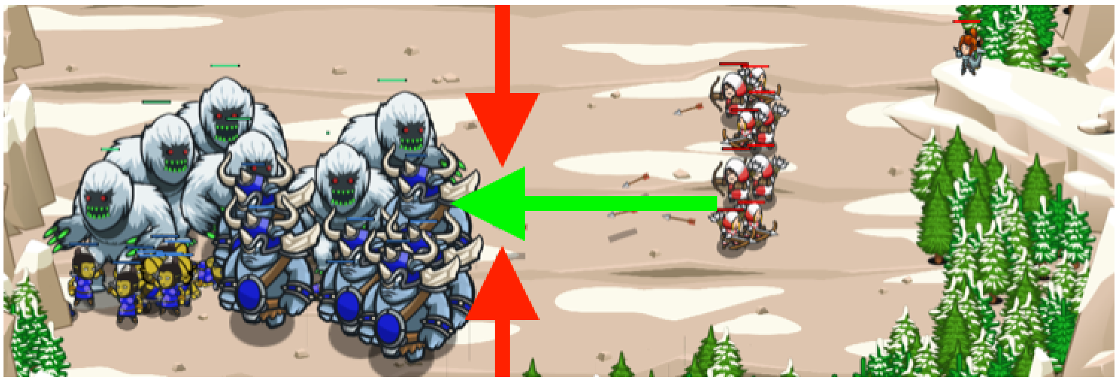
Hunting Party Solution

```
// Command your troops to move east and attack any ogres they see.
// Use for-loops and findFriends.
// You can use findNearestEnemy() on your soldiers to get their nearest enemy instead of yours.
while(true) {
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var friend = friends[i];
        var enemy = friend.findNearestEnemy();
        if(enemy) {
            hero.command(friend, "attack", enemy);
        } else {
            var moveTo = {"x": friend.pos.x + 0.35, "y": friend.pos.y};
            hero.command(friend, "move", moveTo);
        }
    }
}
```

#35. Borrowed Sword

Level Overview and Solutions

Intro



Ogres are fighting yeti.

Use a `for` loop to examine the enemies. Command your archers to attack the enemy with the highest `health` !

When that enemy no longer has the highest health, the archers should switch to the new healthiest enemy!

Default Code

```
// For this level, your hero doesn't fight.
// Command your archers to focus fire on the enemy with the most health!
```

Overview

Good tactics can turn defeat into victory. In this level, ogres are fighting yeti.

If you do nothing, or if you simply target the closest enemy, one side will win and have enough left to defeat your archers.

Instead, use a for-loop to find the enemy with the highest `health` . Then, `command` your archers to attack that enemy.

Hint: Do not continue attacking the toughest enemy until it has been defeated! You want your archers to constantly switch targets to whichever enemy has the highest health at any given moment! This way, whoever wins (between the yeti and ogres) will be weak enough for your archers to finish off.

Borrowed Sword Solution

```
// For this level, your hero doesn't fight.
// Command your archers to focus fire on the enemy with the most health!
while (true) {
    var toughest = null;
    var mostHealth = 0;
    var enemies = hero.findEnemies();
    for(var i=0; i < enemies.length; i++) {
        var enemy = enemies[i];
        if(enemy.health > mostHealth) {
            toughest = enemy;
            mostHealth = enemy.health;
        }
    }

    if(toughest) {
        var friends = hero.findFriends();
        for(var j=0; j < friends.length; j++) {
            var friend = friends[j];
            hero.command(friend, "attack", toughest);
        }
    }
}
```

#36. Summation Summit

Level Overview and Solutions

Intro

Summation Summit is the multiplayer battle arena to test your skills in summoning and commanding troops!

Use what you've learned to earn victory!

Default Code

```
// Defeat the enemy hero in two minutes.

var summonOrder = ["soldier", "soldier"];
var summonCount = 0;

while(true) {
    var enemies = hero.findEnemies();
    var nearestEnemy = hero.findNearest(enemies);

    // Your hero can collect coins and summon troops.
    var summonType = summonOrder[summonCount % summonOrder.length];
    if (hero.gold > hero.costOf(type)) {
        hero.summon(type);
        summonCount += 1;
    }

    // He also commands your allies in battle.
    var friends = hero.findFriends();
    for (var friendIndex = 0; friendIndex < friends.length; ++friendIndex) {
        var friend = friends[friendIndex];
        hero.command(friend, "attack", friend.findNearest(enemies));
    }

    // Gather coins, attack the enemy, or both!

}
```

Overview

Summation Summit

Battle your friends in this head-to-head multiplayer arena! Write the best code and come out on top!

Remember how to summon and command troops:

```
while(true) {
    if(hero.gold > hero.costOf("soldier") {
        hero.summon("soldier");
    }
    var friends = hero.findFriends();
    for(var i = 0; i < friends.length; i++) {
        var friend = friends[i];
        hero.command(friend, "attack", friend.findNearestEnemy());
    }
}
```

There are more tricks to employ, so **optimize** your code and check the **API docs**!

Summation Summit Solution

Coming soon!
