Computer Science 5

JavaScript

Introduces function parameters, function return values and algorithms.

#1. Vital Powers

Level Overview and Solutions

Intro

Break apart vital parts of your code into functions.

Collect gold and summon soldiers while avoiding attacking ogres!

Default Code

```
// This level shows how to define your own functions.
// The code inside a function is not executed immediately. It's saved for later.
// This function has your hero collect the nearest coin.
function pickUpNearestCoin() {
    var items = hero.findItems();
    var nearestCoin = hero.findNearest(items);
    if(nearestCoin) {
        hero.move(nearestCoin.pos);
    }
}

// This function has your hero summon a soldier.
function summonSoldier() {
    // Fill in code here to summon a soldier if you have enough gold.

}

// This function commands your soldiers to attack their nearest enemy.
function commandSoldiers() {
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var enemy = friends[i].findNearestEnemy();
        if(enemy) {
            hero.command(friends[i],"attack", enemy);
        }
    }
}

while(true) {
    // In your loop, you can "call" the functions defined above.
    // The following line causes the code inside the "pickUpNearestCoin" function to be executed.
    pickUpNearestCoin();
    // Call summonSoldier here

    // Call commandSoldiers here

}
```

Overview

Functions are great for breaking up advanced behavior into smaller more "bite-sized" pieces.

In this level, you should collect coins, summon soldiers, and have all soldiers attack their nearest enemy. Each of those actions could be their own function!

Look at the example functions already written for you: `pickUpNearestCoin` and `commandSoldiers` and fill out your own: `summonSoldier` function.

Remember that after **defining** a function, you must **call** it! To **call** a function use `()` at the end of the function's name, like `commandSoldiers()`.

Vital Powers Solution

```
// This level shows how to define your own functions.
// The code inside a function is not executed immediately. It's saved for later.
// This function has your hero collect the nearest coin.
function pickUpNearestCoin() {
    var items = hero.findItems();
    var nearestCoin = hero.findNearest(items);
    if(nearestCoin) {
        hero.move(nearestCoin.pos);
    }
}

// This function has your hero summon a soldier.
function summonSoldier() {
    // Fill in code here to summon a soldier if you have enough gold.
    if(hero.gold >= hero.costOf("soldier")) {
        hero.summon("soldier");
    }
}

// This function commands your soldiers to attack their nearest enemy.
function commandSoldiers() {
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var enemy = friends[i].findNearestEnemy();
        if(enemy) {
            hero.command(friends[i],"attack", enemy);
        }
    }
}

while(true) {
    // In your loop, you can "call" the functions defined above.
    // The following line causes the code inside the "pickUpNearestCoin" function to be executed.
    pickUpNearestCoin();
    // Call summonSoldier here
    summonSoldier();
    // Call commandSoldiers here
    commandSoldiers();
}
```

#2. The Two Flowers

Level Overview and Solutions

Intro

Write a `commandSoldiers` and `pickUpNearestCoin` function to survive the attack!

Remember to call the functions you write inside your `while-true` loop.

Default Code

```
// If the peasant is damaged, the flowers will shrink!

function summonSoldiers() {
    if (hero.gold >= hero.costOf("soldier")) {
        hero.summon("soldier");
    }
}

// Define the function: commandSoldiers

// Define the function: pickUpNearestCoin

var peasant = hero.findByType("peasant")[0];

while(true) {
    summonSoldiers();
    // commandSoldiers();
    // pickUpNearestCoin();

}
```

Overview

For this level, the Two Flowers are connected to the peasant.

If the peasant is damaged, the Two Flowers will take damage and shrink instead of the peasant.

If the peasant is damaged and the Two Flowers are normal sized flowers, the peasant will take damage instead!

When the peasant isn't taking damage, the flowers will grow.

Use your knowledge of commanding units to fill out the `commandSoldiers` and `pickUpNearestCoin` functions.

Remember to uncomment the function calls inside the loop when they're written!

The Two Flowers Solution

```
// If the peasant is damaged, the flowers will shrink!

function summonSoldiers() {
    if (hero.gold >= hero.costOf("soldier")) {
        hero.summon("soldier");
    }
}

// Define the function: commandSoldiers
function commandSoldiers() {
    var soldiers = hero.findByType("soldier");
    for(var i=0; i < soldiers.length; i++) {
        var soldier = soldiers[i];
        var enemy = soldier.findNearestEnemy();
        if(enemy) {
            hero.command(soldier, "attack", enemy);
        }
    }
}

// Define the function: pickUpNearestCoin
function pickUpNearestCoin() {
  var coin = hero.findNearest(hero.findItems());
  if(coin) {
      hero.move(coin.pos);
  }
}

var peasant = hero.findByType("peasant")[0];

while(true) {
    summonSoldiers();
    commandSoldiers();
    pickUpNearestCoin();

}
```

#3. Hunters and Prey

Level Overview and Solutions

Intro

Comamnd a set of troops efficiently!

Archers should only attack enemies that are closer than **25m**.

Soldiers should attack anyone.

Collect coins using a function.

Default Code

```
// Ogres are trying to take out your reindeer!
// Keep your archers back while summoning soldiers to attack.

hero.pickUpCoin = function() {
    // Collect coins.

};

hero.summonTroops = function() {
    // Summon soldiers if you have the gold.

};

// This function has an argument named soldier.
// Arguments are like variables.
// The value of an argument is determined when the function is called.
hero.commandSoldier = function(soldier) {
    // Soldiers should attack enemies.

};

// Write a commandArcher function to tell your archers what to do!
// It should take one argument that will represent the archer passed to the function when it's called.
// Archers should only attack enemies who are closer than 25 meters, otherwise, stay still.

while(true) {
    hero.pickUpCoin();
    hero.summonTroops();
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var friend = friends[i];
        if(friend.type == "soldier") {
            // This friend will be assigned to the variable soldier in commandSoldier
            hero.commandSoldier(friend);
        } else if(friend.type == "archer") {
            // Be sure to command your archers.

        }
    }
}
```

Overview

If your archers move too far away from your reindeer, ogres will ambush the reindeer from the surrounding hills. So, keep the archers back near the reindeer.

In order to do this, you will have to command the archers to `command(archer, "move", archer.pos)` which simply means "stay where you are". You need to do this because the default behavior of your archers, if they haven't received any other commands, is to chase after enemies.

If an enemy is within the archer's attack range (**25 meters**), your archer can safely attack it.

Hunters and Prey Solution

```javascript
// Ogres are trying to take out your reindeer!
// Keep your archers back while summoning soldiers to attack.

function pickUpCoin() {
    // Collect coins.
    var coin = hero.findNearest(hero.findItems());
    if(coin) {
        hero.move(coin.pos);
    }
}

function summonTroops() {
    // Summon soldiers if you have the gold.
    if(hero.costOf("soldier") <= hero.gold) {
        hero.summon("soldier");
    }
}

// This function has an argument named soldier.
// Arguments are like variables.
// The value of an argument is determined when the function is called.
function commandSoldier(soldier) {
    // Soldiers should attack enemies.
    var enemy = soldier.findNearestEnemy();
    if(enemy) {
        hero.command(soldier, "attack", enemy);
    }
}

// Write a commandArcher function to tell your archers what to do!
// It should take one argument that will represent the archer passed to the function when it's called.
// Archers should only attack enemies who are closer than 25 meters, otherwise, stay still.
function commandArcher(archer) {
    var enemy = archer.findNearestEnemy();
    if(enemy && archer.distanceTo(enemy) < 25) {
        hero.command(archer, "attack", enemy);
    } else {
        hero.command(archer, "move", archer.pos);
    }
}

while(true) {
    pickUpCoin();
    summonTroops();
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        var friend = friends[i];
        if(friend.type == "soldier") {
            // This friend will be assigned to the variable soldier in commandSoldier
            commandSoldier(friend);
        } else if(friend.type == "archer") {
            // Be sure to command your archers.
            commandArcher(friend);
        }
    }
}
```

#4. Reaping Fire

Level Overview and Solutions

Intro

The ogres are advancing! Command the flying `"griffin-riders"` to fly over the mines and defend the hero.

Use a series of functions to break each part of the process into easier to understand pieces.

Default Code

```
// The goal is to survive for 30 seconds, and keep the mines intact for at least 30 seconds.

function chooseStrategy() {
    var enemies = hero.findEnemies();
    // If you can summon a griffin-rider, return "griffin-rider"

    // If there is a fangrider on your side of the mines, return "fight-back"

    // Otherwise, return "collect-coins"

}

function commandAttack() {
    // Command your griffin riders to attack ogres.

}

function pickUpCoin() {
    // Collect coins

}

function heroAttack() {
    // Your hero should attack fang riders that cross the minefield.

}

while(true) {
    commandAttack();
    var strategy = chooseStrategy();
    // Call a function, depending on what the current strategy is.

}
```

Overview

Remember, you can find enemy Fangriders with `findByType("fangrider")`, and Griffin Riders with `findByType("griffin-rider")`

Reaping Fire Solution

```
// The goal is to survive for 30 seconds, and keep the mines intact for at least 30 seconds.

function chooseStrategy() {
    var enemies = hero.findEnemies();
    // If you can summon a griffin-rider, return "griffin-rider"
    if(hero.costOf("griffin-rider") <= hero.gold) {
        return "griffin-rider";
    }
    // If there is a fangrider on your side of the mines, return "fight-back"
    var fangriders = hero.findByType("fangrider");
    for(var i = 0; i < fangriders.length; i++) {
        var fangrider = fangriders[i];
        if(fangrider.pos.x < 38) {
            return "fight-back";
        }
    }
    // Otherwise, return "collect-coins"
    return "collect-coins";
}

function commandAttack() {
    // Command your griffin riders to attack ogres.
    var griffins = hero.findFriends();
    var enemies = hero.findEnemies();
    for(var i = 0; i < griffins.length; i++) {
        var griffin = griffins[i];
        var enemy = griffin.findNearest(enemies);
        if(enemy) {
            hero.command(griffin, "attack", enemy);
        }
    }
}

function pickUpCoin() {
    // Collect coins
    var items = hero.findItems();
    var item = hero.findNearest(items);
    if(item) {
        hero.move(item.pos);
    }
}

function heroAttack() {
    // Your hero should attack fang riders that cross the minefield.
    var enemy = hero.findNearest(hero.findByType("fangrider"));
    if(enemy && hero.distanceTo(enemy) < 15) {
        hero.attack(enemy);
    }
}

while(true) {
    commandAttack();
    var strategy = chooseStrategy();
    // Call a function, depending on what the current strategy is.
    if(strategy == "griffin-rider") {
        hero.summon("griffin-rider");
    } else if (strategy == "fight-back") {
        heroAttack();
    } else if (strategy == "collect-coins") {
        pickUpCoin();
    }
}
```

#5. Toil and Trouble

Level Overview and Solutions

Intro

Soldiers and archers should target different units to complete this level.

`"soldier"` s should attack the `"witch"` .

`"archer"` s should attack their nearest enemy.

Default Code

```
// Ogre Witches have some unpleasant surprises ready for you.

// Define a chooseTarget function which takes a friend argument
// Returns the a target to attack, depending on the type of friend.
// Soldiers should attack the witches, archers should attack nearest enemy.


while(true) {
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        // Use your chooseTarget function to decide what to attack.

    }
}
```

Overview

Define a `chooseTarget` function that accepts an argument called `friend` .

Find the `nearest` enemy to the friend, as well as the nearest enemy with type `"witch"` *(use `findByType`)*

Your soldiers should target witches first, or if there are no witches, then the nearest enemy. Other units should always target the nearest enemy.

Hint: Each soldier should attack the `"witch"` nearest to that soldier. This will split your soldiers into two groups, and they won't get injured by the splash damage done by the witches' attacks.

Toil and Trouble Solution

```
// Ogre Witches have some unpleasant surprises ready for you.

// Define a chooseTarget function which takes a friend argument
// Returns the a target to attack, depending on the type of friend.
// Soldiers should attack the witches, archers should attack nearest enemy.
function chooseTarget(friend) {
    var target = null;
    if(friend.type == "archer") {
        var enemies = hero.findEnemies();
        target = friend.findNearest(enemies);
    } else if (friend.type == "soldier") {
        var witches = hero.findByType("witch");
        target = friend.findNearest(witches);
    }
    return target;
}

while(true) {
    var friends = hero.findFriends();
    for(var i=0; i < friends.length; i++) {
        // Use your chooseTarget function to decide what to attack.
        var friend = friends[i];
        var enemy = chooseTarget(friend);
        if(enemy) {
            hero.command(friend, "attack", enemy);
        }
    }
}
```

#6. Mixed Unit Tactics

Level Overview and Solutions

Intro

An open-ended defense level!

Survive for 30 seconds and collect 300 gold using your learned skills.

Can you make it to 60 seconds?

Default Code

```
// Practice using modulo to loop over an array

// Choose the mix and order of units you want to summon by populating this array:
var summonTypes = [];

function summonTroops() {
    // Use % to wrap around the summonTypes array based on this.built.length
    //var type = summonTypes[???];
    hero.say("I should summon troops!");
}
```

Overview

First, fill in the `summonTypes` array with the types of units you'd like to summon, in order.

Then, complete the `summonTroops` function using

```
hero.built.length % summonTypes.length
```

to loop over the `summonTypes` array.

You'll also need to create a `collectCoins` function and a `commandTroops` function.

Tip: In `commandTroops` you will want to skip over any friends who have `type == "palisade"` !

Mixed Unit Tactics Solution

```
// Practice using modulo to loop over an array

// Choose the mix and order of units you want to summon by populating this array:
var summonTypes = ["soldier", "archer"];

function summonTroops() {
    // Use % to wrap around the summonTypes array based on this.built.length
    var type = summonTypes[hero.built.length % summonTypes.length];
    if(hero.costOf(type) <= hero.gold) {
        hero.summon(type);
    }
}
function gatherCoins() {
    var items = hero.findItems();
    var item = hero.findNearest(items);
    if(item) {
        hero.move(item.pos);
    }
}
function commandTroops() {
    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);
        }
    }
}
while(true) {
    summonTroops();
    gatherCoins();
    commandTroops();
}
```

#7. Steelclaw Gap

Level Overview and Solutions

Intro

Use the modulo operator (`%`) to send two units to each guard point.

You need to summon 8 units and move to 4 points, so `unitIndex % 4` is the requried `defendIndex` .

Default Code

```
// This level introduces the % operator, also known as the modulo operator.
// a % b returns the remainder of a divided by b
// This can be used to wrap around to the beginning of an array when an index might be greater than the
    length

var defendPoints = [{"x": 35, "y": 63},{"x": 61, "y": 63},{"x": 32, "y": 26},{"x": 64, "y": 26}];

var summonTypes = ["soldier","soldier","soldier","soldier","archer","archer","archer","archer"];

// You start with 360 gold to build a mixture of soldiers and archers.
// this.built is an array of the troops you have built, ever.
// Here we use "this.built.length % summonTypes.length" to wrap around the summonTypes array
function summonTroops() {
    var type = summonTypes[hero.built.length % summonTypes.length];
    if(hero.gold >= hero.costOf(type)) {
        hero.summon(type);
    }
}

function commandTroops() {
    var friends = hero.findFriends();
    for(var friendIndex=0; friendIndex < friends.length; friendIndex++) {
        var friend = friends[friendIndex];
        // Use % to wrap around defendPoints based on friendIndex

        // Command your minion to defend the defendPoint

    }
}

while(true) {
    summonTroops();
    commandTroops();
}
```

Overview

The % operator is known as the modulo operator.

`a % b` gives you the remainder (as an integer) of `a / b` . So `12 % 5 == 2` .

This can be used to wrap around an array, for example:

Using the array: `summonTypes = ["soldier","archer","peasant","paladin"]`

With: `type = summonTypes[ i % summonTypes.length ]`

`0 % 4 == 0` so `type == "soldier"`

`1 % 4 == 1` so `type == "archer"`

`2 % 4 == 2` so `type == "peasant"`

`3 % 4 == 3` so `type == "paladin"`

`4 % 4 == 0` so `type == "soldier"`

`5 % 4 == 1` so `type == "archer"`

etc...

Steelclaw Gap Solution

```
// This level introduces the % operator, also known as the modulo operator.
// a % b returns the remainder of a divided by b
// This can be used to wrap around to the beginning of an array when an index might be greater than the
    length

var defendPoints = [{"x": 35, "y": 63},{"x": 61, "y": 63},{"x": 32, "y": 26},{"x": 64, "y": 26}];

var summonTypes = ["soldier","soldier","soldier","soldier","archer","archer","archer","archer"];

// You start with 360 gold to build a mixture of soldiers and archers.
// this.built is an array of the troops you have built, ever.
// Here we use "this.built.length % summonTypes.length" to wrap around the summonTypes array
function summonTroops() {
    var type = summonTypes[hero.built.length % summonTypes.length];
    if(hero.gold >= hero.costOf(type)) {
        hero.summon(type);
    }
}

function commandTroops() {
    var friends = hero.findFriends();
    for(var friendIndex=0; friendIndex < friends.length; friendIndex++) {
        var friend = friends[friendIndex];
        // Use % to wrap around defendPoints based on friendIndex
        var point = defendPoints[friendIndex % defendPoints.length];
        // Command your minion to defend the defendPoint
        hero.command(friend, "defend", point);
    }
}

while(true) {
    summonTroops();
    commandTroops();
}
```

#8. Ring Bearer

Level Overview and Solutions

Intro

Use the `findSoldierOffset` function to find a soldier's radial offset.

Add that number to the peasant's position to create a circular protection area around the ring-bearing peasant!

Default Code

```
// You must escort a powerful magical ring back to town to be studied.
// The goal is to escape, not fight. More ogres lurk in the surrounding mountains!
// Make a circle of soldiers around the peasant!
// We give you two functions to help with this:

// findSoldierOffset figures out the position a soldier should stand at in relation to the peasant.
// The first argument 'soldiers' should be an array of your soldiers.
// The second argument 'i' is the index of the soldier (in soldiers) you want to find the position for.
function findSoldierOffset(soldiers, i) {
    var soldier = soldiers[i];
    var angle = i * 360 / soldiers.length;
    return radialToCartesian(5, angle);
}

// This function does the math to determine the offset a soldier should stand at.
function radialToCartesian(radius, degrees) {
    var radians = Math.PI / 180 * degrees;
    var xOffset = radius * Math.cos(radians);
    var yOffset = radius * Math.sin(radians);
    return {x: xOffset, y: yOffset};
}

var peasant = hero.findByType("peasant")[0];

// Use findByType to get an array of your soldiers.
while(true) {
    // Use a for-loop to iterate over your array of soldiers.

    // Find the offset for a soldier.
    // Add the offset.x and offset.y to the peasant's pos.x and pos.y.
    // Command the soldier to move to the new offset position.

    // The hero should keep pace with the peasant!
    hero.move({x: hero.pos.x + 0.2, y: hero.pos.y});
}
```

Overview

For this level, you should use the functions that we supplied to calculate where a soldier should move to, in order to form a ring of soldiers around the peasant.

You don't have to fully understand what those functions do to use them, other than the inputs and outputs of `findSoldierOffset(soldiers, i)`.

The first argument `soldiers` is the array containing all of your soldiers.

The second argument `i` is the index of the soldier (in the soldiers array) that you want to find the offset position for.

The `findSoldierOffset` function returns an `{x, y}` object containing the offset position the soldier at position `i` should stand, relative to the peasant.

To calculate where the soldier should move to, you add the offset to the peasant's position like this:

```
moveTo = {"x": peasant.pos.x + offset.x, "y": peasant.pos.y + offset.y}
command(soldier, "move", moveTo)
```

Ring Bearer Solution

```
// You must escort a powerful magical ring back to town to be studied.
// The goal is to escape, not fight. More ogres lurk in the surrounding mountains!
// Make a circle of soldiers around the peasant!
// We give you two functions to help with this:

// findSoldierOffset figures out the position a soldier should stand at in relation to the peasant.
// The first argument 'soldiers' should be an array of your soldiers.
// The second argument 'i' is the index of the soldier (in soldiers) you want to find the position for.
function findSoldierOffset(soldiers, i) {
    var soldier = soldiers[i];
    var angle = i * 360 / soldiers.length;
    return radialToCartesian(5, angle);
}

// This function does the math to determine the offset a soldier should stand at.
function radialToCartesian(radius, degrees) {
    var radians = Math.PI / 180 * degrees;
    var xOffset = radius * Math.cos(radians);
    var yOffset = radius * Math.sin(radians);
    return {x: xOffset, y: yOffset};
}

var peasant = hero.findByType("peasant")[0];

// Use findByType to get an array of your soldiers.
var soldiers = hero.findByType("soldier");
while(true) {
    // Use a for-loop to iterate over your array of soldiers.
    for(var i=0; i < soldiers.length; i++) {
        // Find the offset for a soldier.
        var offset = findSoldierOffset(soldiers, i);
        // Add the offset.x and offset.y to the peasant's pos.x and pos.y.
        var x = peasant.pos.x + offset.x;
        var y = peasant.pos.y + offset.y;
        var moveTo = { "x": x, "y": y };
        // Command the soldier to move to the new offset position.
        hero.command(soldiers[i], "move", moveTo);
    }
    // The hero should keep pace with the peasant!
    hero.move({x: hero.pos.x + 0.2, y: hero.pos.y});
}
```

#9. Library Tactician

Level Overview and Solutions

Intro

Write a function to return the strongest enemy, and command your archers to attack it!

Hushbaum will provide healing support throughout the battle.

Default Code

```
// Hushbaum has been ambushed by ogres!
// She is busy healing her soldiers, you should command them to fight!
// The ogres will send more troops if they think they can get to Hushbaum or your archers, so keep them
    inside the circle!

// Soldiers spread out in a circle and defend.
function commandSoldier(soldier, soldierIndex, numSoldiers) {
    var angle = Math.PI * 2 * soldierIndex / numSoldiers;
    var defendPos = {x: 41, y: 40};
    defendPos.x += 10 * Math.cos(angle);
    defendPos.y += 10 * Math.sin(angle);
    hero.command(soldier, "defend", defendPos);
}

// Find the strongest target (most health)
// This function returns something! When you call the function, you will get some value back.
function findStrongestTarget() {
    var mostHealth = 0;
    var bestTarget = null;
    var enemies = hero.findEnemies();
    // Figure out which enemy has the most health, and set bestTarget to be that enemy.

    // Only focus archers' fire if there is a big ogre.
    if (bestTarget && bestTarget.health > 15) {
        return bestTarget;
    } else {
        return null;
    }
}


// If the strongestTarget has more than 15 health, attack that target. Otherwise, attack the nearest
    target.
function commandArcher(archer) {
    var nearest = archer.findNearestEnemy();
    if(archerTarget) {
        hero.command(archer, "attack", archerTarget);
    } else if(nearest) {
        hero.command(archer, "attack", nearest);
    }
}

var archerTarget = null;
while(true) {
    // If archerTarget is defeated or doesn't exist, find a new one.
    if(!archerTarget || archerTarget.health <= 0) {
        // Set archerTarget to be the target that is returned by findStrongestTarget()
        archerTarget = findStrongestTarget();
    }
    var friends = hero.findFriends();
    var soldiers = hero.findByType("soldier");
    // Create a variable containing your archers.

    for(var i=0; i < soldiers.length; i++) {
        var soldier = soldiers[i];
        commandSoldier(soldier, i, soldiers.length);
    }
    // use commandArcher() to command your archers

}
```

Overview

Functions can *return* a value.

In this level, you have `findStrongestTarget()` which should locate and return the enemy with the highest health.

When a function returns a value, you can assign that value to a variable when you call the function, like: `archerTarget = findStrongestTarget()` (this may vary a bit depending on the language you use, but the idea is the same).

Library Tactician Solution

```
// Hushbaum has been ambushed by ogres!
// She is busy healing her soldiers, you should command them to fight!
// The ogres will send more troops if they think they can get to Hushbaum or your archers, so keep them
    inside the circle!

// Soldiers spread out in a circle and defend.
function commandSoldier(soldier, soldierIndex, numSoldiers) {
    var angle = Math.PI * 2 * soldierIndex / numSoldiers;
    var defendPos = {x: 41, y: 40};
    defendPos.x += 10 * Math.cos(angle);
    defendPos.y += 10 * Math.sin(angle);
    hero.command(soldier, "defend", defendPos);
}

// Find the strongest target (most health)
// This function returns something! When you call the function, you will get some value back.
function findStrongestTarget() {
    var mostHealth = 0;
    var bestTarget = null;
    var enemies = hero.findEnemies();
    // Figure out which enemy has the most health, and set bestTarget to be that enemy.
    for(var i=0; i < enemies.length; i++) {
        var enemy = enemies[i];
        if(enemy.health > mostHealth) {
            bestTarget = enemy;
            mostHealth = enemy.health;
        }
    }
    // Only focus archers' fire if there is a big ogre.
    if (bestTarget && bestTarget.health > 15) {
        return bestTarget;
    } else {
        return null;
    }
}


// If the strongestTarget has more than 15 health, attack that target. Otherwise, attack the nearest
    target.
function commandArcher(archer) {
    var nearest = archer.findNearestEnemy();
    if(archerTarget) {
        hero.command(archer, "attack", archerTarget);
    } else if(nearest) {
        hero.command(archer, "attack", nearest);
    }
}

var archerTarget = null;
while(true) {
    // If archerTarget is defeated or doesn't exist, find a new one.
    if(!archerTarget || archerTarget.health <= 0) {
        // Set archerTarget to be the target that is returned by findStrongestTarget()
        archerTarget = findStrongestTarget();
    }
    var friends = hero.findFriends();
    var soldiers = hero.findByType("soldier");
    // Create a variable containing your archers.
    var archers = hero.findByType("archer");
    for(var i=0; i < soldiers.length; i++) {
        var soldier = soldiers[i];
        commandSoldier(soldier, i, soldiers.length);
    }
    // use commandArcher() to command your archers
    for(i=0; i < archers.length; i++) {
        var archer = archers[i];
        commandArcher(archer);
    }
}
```

#10. The Geometry of Flowers

Level Overview and Solutions

Intro

Draw a `circle` and a `square` then draw whatever you like!

Default Code

```
// You now have the Ring of Flowers! You can do:
// toggleFlowers(true/false) - turns flowers on or off.
// setFlowerColor("random") - can also be "pink", "red", "blue", "purple", "yellow", or "white".

// Here are some functions for drawing shapes:
// x, y - center of the shape
// size - size of the shape (radius, side length)
hero.drawCircle = function(x, y, size) {
    var angle = 0;
    hero.toggleFlowers(false);
    while (angle <= Math.PI * 2) {
        var newX = x + (size * Math.cos(angle));
        var newY = y + (size * Math.sin(angle));
        hero.moveXY(newX, newY);
        hero.toggleFlowers(true);
        angle += 0.2;
    }
};

hero.drawSquare = function(x, y, size) {
    hero.toggleFlowers(false);
    var cornerOffset = size / 2;
    hero.moveXY(x - cornerOffset, y - cornerOffset);
    hero.toggleFlowers(true);
    hero.moveXY(x + cornerOffset, y - cornerOffset);
    hero.moveXY(x + cornerOffset, y + cornerOffset);
    hero.moveXY(x - cornerOffset, y + cornerOffset);
    hero.moveXY(x - cornerOffset, y - cornerOffset);
};

var redX = {x: 28, y: 36};
var whiteX = {x: 44, y: 36};

// Pick a color.

// Draw a size 10 circle at the redX.

// Change the color!

// Draw a size 10 square at the whiteX.

// Now experiment with drawing whatever you want!
```

Overview

The sample code gives you functions for `drawCircle` and `drawSquare`.

To pass the level, you need to draw a size `10` circle at the coordinates given for `redX` and a size `10` square at the coordinates for `whiteX`.

After that, you can experiment with drawing other shapes if you like, or move on to the **Mountain Flower Grove** to practice your drawing skills!

The Geometry of Flowers Solution

```
// You now have the Ring of Flowers! You can do:
// toggleFlowers(true/false) - turns flowers on or off.
// setFlowerColor("random") - can also be "pink", "red", "blue", "purple", "yellow", or "white".

// Here are some functions for drawing shapes:
// x, y - center of the shape
// size - size of the shape (radius, side length)
hero.drawCircle = function(x, y, size) {
    var angle = 0;
    hero.toggleFlowers(false);
    while (angle <= Math.PI * 2) {
        var newX = x + (size * Math.cos(angle));
        var newY = y + (size * Math.sin(angle));
        hero.moveXY(newX, newY);
        hero.toggleFlowers(true);
        angle += 0.2;
    }
};

hero.drawSquare = function(x, y, size) {
    hero.toggleFlowers(false);
    var cornerOffset = size / 2;
    hero.moveXY(x - cornerOffset, y - cornerOffset);
    hero.toggleFlowers(true);
    hero.moveXY(x + cornerOffset, y - cornerOffset);
    hero.moveXY(x + cornerOffset, y + cornerOffset);
    hero.moveXY(x - cornerOffset, y + cornerOffset);
    hero.moveXY(x - cornerOffset, y - cornerOffset);
};

var redX = {x: 28, y: 36};
var whiteX = {x: 44, y: 36};

// Pick a color.
hero.setFlowerColor("red");

// Draw a size 10 circle at the redX.
hero.drawCircle(redX.x, redX.y, 10);

// Change the color!
hero.setFlowerColor("purple");

// Draw a size 10 square at the whiteX.
hero.drawSquare(whiteX.x, whiteX.y, 10);

// Now experiment with drawing whatever you want!
```

#11. The Spy Among Us

Level Overview and Solutions

Intro

A spy is among us!

Find the friend with the letter `z` to find the spy!

Default Code

```
// The inner gate can hold for a long time.
// However, one of these peasants is an OGRE SPY!
// There is a hint! The spy's name has the letter "z"

// This function checks for a specific letter in a word.
// A string is just an array! Loop over it like an array
function letterInWord(word, letter) {
    for(var i=0; i < word.length; i++) {
        var character = word[i];
        // If character is equal to letter, return true

    }
    // The letter isn't in the word, so return false
    return false;
}

var spyLetter = "z";
var friends = hero.findFriends();

for (var j = 0; j < friends.length; j++) {
    var friendName = friends[j].id;
    if (letterInWord(friendName, spyLetter)) {
        // Reveal the spy!
        hero.say(friendName + " is a spy!!!");
    } else {
        hero.say(friendName + " is a friend.");
    }
}
```

Overview

Strings are like arrays. They have a length and can be index-referenced similar to arrays:

```
hero.say("Hello World"[0]); // Hero says 'H'
hero.say("Hello World".length); // Hero says 11
```

In this level you'll want to iterate over the length of the array to find if any of the characters inside of a friend's name are the same as the spy character!

The Spy Among Us Solution

```
// The inner gate can hold for a long time.
// However, one of these peasants is an OGRE SPY!
// There is a hint! The spy's name has the letter "z"

// This function checks for a specific letter in a word.
// A string is just an array! Loop over it like an array
function letterInWord(word, letter) {
    for(var i=0; i < word.length; i++) {
        var character = word[i];
        // If character is equal to letter, return true
        if(character == letter) {
            return true;
        }
    }
    // The letter isn't in the word, so return false
    return false;
}

var spyLetter = "z";
var friends = hero.findFriends();

for (var j = 0; j < friends.length; j++) {
    var friendName = friends[j].id;
    if (letterInWord(friendName, spyLetter)) {
        // Reveal the spy!
        hero.say(friendName + " is a spy!!!");
    } else {
        hero.say(friendName + " is a friend.");
    }
}
```

#12. In My Name

Level Overview and Solutions

Intro

Find the correct chest by detecting an anomaly in Thoktar's name!

Choose poorly, and the skeletons will claim their own prize!

Default Code

```
// You must to find the treasure.
// Use Thoktar's name as a hint.
// The correct chest is the same index as "z".


// This function should return the index of a letter:
function letterIndex(word, letter) {
    // Step through each letter as an index of the word.

        // Store a character from the word with the current index.

        // If it is the required letter:

            // Then return the current index (number).

    // If nothing, return a default value
    return -1;
}

var ogreLetter = "z";
var shaman = hero.findByType("thoktar")[0];

// Find the index and use it for finding the treasure.
var chestIndex = letterIndex(shaman.id, ogreLetter);
hero.moveXY(16 + chestIndex * 8, 36);
```

Overview

Find the treasure chest which contains the treasure!

Thoktar's name holds the secret. Whichever index in Thoktar's name is `"z"`, is the index of chests that the player should approach.

Iterate over each element of a string with the following:

```
var string = "Hello, World!";
for(var i = 0; i < string.length; i++) {
    var letter = string[i];
    hero.say(letter); // The hero will say each letter individually.
}
```

Use this knowledge to defeat Thoktar!

In My Name Solution

```
// You must to find the treasure.
// Use Thoktar's name as a hint.
// The correct chest is the same index as "z".


// This function should return the index of a letter:
function letterIndex(word, letter) {
    // Step through each letter as an index of the word.
    for(var i = 0; i < word.length; i++) {
        // Store a character from the word with the current index.
        var char = word[i];
        // If it is the required letter:
        if(char == letter) {
            // Then return the current index (number).
            return i;
        }
    }
    // If nothing, return a default value
    return -1;
}

var ogreLetter = "z";
var shaman = hero.findByType("thoktar")[0];

// Find the index and use it for finding the treasure.
var chestIndex = letterIndex(shaman.id, ogreLetter);
hero.moveXY(16 + chestIndex * 8, 36);
```

#13. Highlanders

Level Overview and Solutions

Intro

Those weird statues near the gates contain the blackest magic!

Pass through it without a shield and you will lose your soldiers.

Only send forth the magic-immune highlanders! Search through your friend's names to find which ones are highlanders.

Default Code

```
// You must defeat the ogres
// But they are using black magic!
// Only the highlander soldiers are immune.
// Find highlanders, their names always contain "mac"

var highlanderName = "mac";

// This function should search for a string inside of a word:
function wordInString(string, word) {
    var lenString = string.length;
    var lenWord = word.length;
    // Step through indexes (i) from 0 to (lenString - lenWord)

        // For each of them through indexes (j) of the word length

            // If [i + j]th letter of the string is not equal [j]th letter of world, then break loop

            // if [j]th is the last letter of the word (j == lenWord - 1), then return True.

    // If loops are ended then the word is not inside the string. Return False.

    return true; // Δ Remove this when the function is written.
}

// Look at your soldiers and choose highlanders only
var soldiers = hero.findFriends();
for (var i = 0; i < soldiers.length; i++) {
    var soldier = soldiers[i];
    if (wordInString(soldier.id, highlanderName)) {
        hero.say(soldier.id + " be ready.");
    }
}

//
hero.say("ATTACK!!!");
```

Overview

In this level you need to only send forward soldiers with `"mac"` in their name.

Searching through strings is tough work, but consider each part step-by-step! Programmtically, if you will:

```
var haystack = "Hello, world!";
var needle = "o, w";
var needleIndex = 0;
for(var i = 0; i < haystack.length; i++) {
    var letter = haystack[i];
    // Check the letter matches the letter at the current needleIndex
    if(letter === needle[needleIndex]) {
        // Increment the index by one so in the next iteration of the for-loop
        // This checks if the substring is still valid on a letter-by-letter basis.
        needleIndex += 1;
        // Check if the needle index is greater than needle length
        if(needleIndex >= needle.length) {
            // Since we've gone through each index of our needle string, we know it exists!
            hero.say("Yep! " + needle + " is in " + haystack + "!");
            break;
        }
    } else {
        // Reset the index because the potential sub-string no longer matches
        needleIndex = 0;
    }
}
```

Highlanders Solution

```
// You must defeat the ogres
// But they are using black magic!
// Only the highlander soldiers are immune.
// Find highlanders, their names always contain "mac"

var highlanderName = "mac";

// This function should search for a string inside of a word:
function wordInString(string, word) {
    var lenString = string.length;
    var lenWord = word.length;
    // Step through indexes (i) from 0 to (lenString - lenWord)
    for(var i = 0; i < lenString - lenWord; i++) {
        // For each of them through indexes (j) of the word length
        for(var j = 0; j < lenWord; j++) {
            // If [i + j]th letter of the string is not equal [j]th letter of world, then break loop
            if(string[i+j] != word[j]) {
                break;
            }
            // if [j]th is the last letter of the word (j == lenWord - 1), then return True.
            else if(j === lenWord - 1) {
                return true;
            }
        }
    }
    // If loops are ended then the word is not inside the string. Return False.
    return false;
}

// Look at your soldiers and choose highlanders only
var soldiers = hero.findFriends();
for (var i = 0; i < soldiers.length; i++) {
    var soldier = soldiers[i];
    if (wordInString(soldier.id, highlanderName)) {
        hero.say(soldier.id + " be ready.");
    }
}

//
hero.say("ATTACK!!!");
```

#14. Perimeter Defense

Level Overview and Solutions

Intro

Set up the town with a formidable defense!

Patrol the perimeter of the village and tell peasants to build at each spot.

Default Code

```
// We need to build guard towers around the village.
// Each peasant can build one tower.
// Show them the place to build.
// These towers are automatic and will attack ALL units outside the town.

// First move along the north border (y=60) from x=40 to x=80 with the step 20.
for (var x = 40; x <= 80; x += 20) {
    // Move at each point and say anything.
    hero.moveXY(x, 60);
    hero.say("Here");
}
// Next move along the east border (x=80) from y=40 to y=20 with the negative step -20.
for (var y = 40; y >= 20; y -= 20) {
    hero.moveXY(80, y);
    hero.say("Here");
}
// Continue for the two remaining borders.
// Next the south border (y=20) from x=60 to x=20 with the negative step -20.

// Next the west border (x=20) from y=40 to y=60 with the step 20.

// Don't forget hide inside the village.
hero.moveXY(50, 40);
```

Overview

`for-loops` can be used for more than iterating over arrays! It is possible to increment by 2, 4, or even 10 or 20 units at a time!

```
for(var i = 0; i < 100; i += 10) {
    hero.say(i); // The hero will say 0, 10, 20,... etc!
}
```

It is also possible to change the starting position:

```
for(var i = 20; i < 100; i += 20) {
    hero.say(i); // The hero will say 20, 40, 60,... etc!
}
```

Use this to navigate around the perimeter of the town and use `say` to tell the peasants when to build.

Perimeter Defense Solution

```
// We need to build guard towers around the village.
// Each peasant can build one tower.
// Show them the place to build.
// These towers are automatic and will attack ALL units outside the town.

// First move along the north border (y=60) from x=40 to x=80 with the step 20.
for (var x = 40; x <= 80; x += 20) {
    // Move at each point and say anything.
    hero.moveXY(x, 60);
    hero.say("Here");
}
// Next move along the east border (x=80) from y=40 to y=20 with the negative step -20.
for (var y = 40; y >= 20; y -= 20) {
    hero.moveXY(80, y);
    hero.say("Here");
}
// Continue for the two remaining borders.
// Next the south border (y=20) from x=60 to x=20 with the negative step -20.
for (x = 60; x >= 20; x -=20) {
    hero.moveXY(x, 20);
    hero.say("Here");
}
// Next the west border (x=20) from y=40 to y=60 with the step 20.
for (y = 40; y <= 60; y += 20) {
    hero.moveXY(20, y);
    hero.say("Here");
}
// Don't forget hide inside the village.
hero.moveXY(50, 40);
```

#15. Dangerous Tracks

Level Overview and Solutions

Intro

There are many entrances that lead into the village. Protect the village and mine all the entrances.

You have 80 seconds before an ogre hunter squad arrives.

Use `for-loop` s to save time and make your code cleaner.

Default Code

```
// Protect the village with fire traps.
// Mine all passages in four directions.
// You have 80 seconds before the ogres attack.

// Build traps on the line y=114 from x=40 to x=112 with step=24.
function buildNorthLine() {
    for (var x = 40; x <= 112; x += 24) {
        hero.buildXY("fire-trap", x, 114);
    }
}

// Build traps on the line x=140 from y=110 to y=38 with step=18.
function buildEastLine() {
    // Complete this function:

}

// Build traps on the line y=22 from x=132 to x=32 with step=20.
function buildSouthLine() {
    // Complete this function:

}

// Build traps on the line x=20 from y=28 to y=108 with step=16.
function buildWestLine() {
    // Complete this function:

}

buildNorthLine();
buildEastLine();
buildSouthLine();
buildWestLine();
hero.moveXY(40, 94);
```

Overview

Remember how to advance over `for-loop` with numbers higher 1:

```
for(var i = 25; i < 100; i += 5) {
    hero.moveXY(i, 50); // Move from 25, 50 to 95, 50 with 5-unit steps.
}
```

Dangerous Tracks Solution

```
// Protect the village with fire traps.
// Mine all passages in four directions.
// You have 80 seconds before the ogres attack.

// Build traps on the line y=114 from x=40 to x=112 with step=24.
function buildNorthLine() {
    for (var x = 40; x <= 112; x += 24) {
        hero.buildXY("fire-trap", x, 114);
    }
}

// Build traps on the line x=140 from y=110 to y=38 with step=18.
function buildEastLine() {
    // Complete this function:
    for (var y = 110; y >= 38; y -= 18) {
        hero.buildXY("fire-trap", 140, y);
    }
}

// Build traps on the line y=22 from x=132 to x=32 with step=20.
function buildSouthLine() {
    // Complete this function:
    for (var x = 132; x >= 32; x -= 20) {
        hero.buildXY("fire-trap", x, 22);
    }
}

// Build traps on the line x=20 from y=28 to y=108 with step=16.
function buildWestLine() {
    // Complete this function:
    for (var y = 28; y <= 108; y += 16) {
        hero.buildXY("fire-trap", 20, y);
    }
}

buildNorthLine();
buildEastLine();
buildSouthLine();
buildWestLine();
hero.moveXY(40, 94);
```

#16. Resource Valleys

Level Overview and Solutions

Intro

Command peasants to pick their nearest coin, but, don't let them target the wrong coin!

Filter the coins by value and have each peasant only find the nearest of special arrays.

Default Code

```
// Collect all the coins!
// The peasants are unable to get the coins from other areas
// However, each area only spawns a certain value of coin!
// Filter through all the items and command the peasants accordingly.

function commandPeasant(peasant, coins) {
    // Command the peasant to the nearest of their array

}

var friends = hero.findFriends();
var peasants = {
    "Aurum":friends[0],
    "Argentum":friends[1],
    "Cuprum":friends[2]
};

while(true) {
    var items = hero.findItems();
    var goldCoins = [];
    var silverCoins = [];
    var bronzeCoins = [];
    for(var i = 0; i < items.length; i++) {
        var item = items[i];
        if(item.value == 3) {
            goldCoins.push(item);
        }
        // Put bronze and silver coins in their approriate array:

    }
    commandPeasant(peasants.Aurum, goldCoins);
    commandPeasant(peasants.Argentum, silverCoins);
    commandPeasant(peasants.Cuprum, bronzeCoins);
}
```

Overview

In this level, the hero will need to filter coins so there won't be any confusion amongst the peasants!

Filtering is the process of keeping or removing certain elements. Specifically in this level, the hero should filter each of the coins into their own array to pass to the corresponding peasant.

Another way of doing what is demonstrated in the level is as follows:

```
// Take an array with lots of elements
var enemies = hero.findEnemies();
// Set aside a new array to put our desired types.
var throwers = [];
// Iterate over all the elements our searchable array.
for(var i = 0; i < enemies.length; i++) {
    enemy = enemies[i];
    // Check if the element matches our desired condition.
    if(enemy.type == "thrower") {
        // If so, push that element into our subset array.
        throwers.push(enemy);
    }
}
// Now we have an array of all the throwers!
throwers;
```

Resource Valleys Solution

```
// Collect all the coins!
// The peasants are unable to get the coins from other areas
// However, each area only spawns a certain value of coin!
// Filter through all the items and command the peasants accordingly.

function commandPeasant(peasant, coins) {
    // Command the peasant to the nearest of their array
    var coin = peasant.findNearest(coins);
    if(coin) {
        hero.command(peasant, "move", coin.pos);
    }
}

var friends = hero.findFriends();
var peasants = {
    "Aurum":friends[0],
    "Argentum":friends[1],
    "Cuprum":friends[2]
};

while(true) {
    var items = hero.findItems();
    var goldCoins = [];
    var silverCoins = [];
    var bronzeCoins = [];
    for(var i = 0; i < items.length; i++) {
        var item = items[i];
        if(item.value == 3) {
            goldCoins.push(item);
        }
        // Put bronze and silver coins in their approriate array:
        else if(item.value == 2) {
            silverCoins.push(item);
        }
        else if(item.value == 1) {
            bronzeCoins.push(item);
        }
    }
    commandPeasant(peasants.Aurum, goldCoins);
    commandPeasant(peasants.Argentum, silverCoins);
    commandPeasant(peasants.Cuprum, bronzeCoins);
}
```

#17. Flawless Pairs

Level Overview and Solutions

Intro

You need to collect 4 pairs of gems to escape from the cemetry.

Each pair must be two gems of the same value. If you make a mistake, you will faint!

The wizard helps you with the "Haste" spell. Just return to the start point after each collected gem.

Default Code

```
// Collect 4 pairs of gems.
// Each pair must contain equal valued gems.

// This function returns two items with the same value.
function findValuePair(items){
    // Check each possible pair in the array.
    // Iterate indexes 'i' from 0 to the last one.
    for (var i = 0; i < items.length; i++) {
        var itemI = items[i];
        // Iterate indexes 'j' from 0 to the last.
        for (var j = 0; j < items.length; j++) {
            // If it's the same element, then skip it.
            if (i == j) {
                continue;
            }
            var itemJ = items[j];
            // If we found a pair with two equal gems, then return them.
            if (itemI.value === itemJ.value) {
                return [itemI, itemJ];
            }
        }
    }
    // Return an empty array if no pair exists.
    return null;
}

while (true) {
    var gems = hero.findItems();
    var gemPair = findValuePair(gems);
    // If the gemPair exists, collect the gems!
    if (gemPair) {
        var gemA = gemPair[0];
        var gemB = gemPair[1];
        // Move to the first gem.

        // Return to get the haste from the wizard.

        // Then move to the second gem.

        // Return to get the haste from the wizard.

    }
}
```

Overview

In this level you must search through an array and find two elements that are similar.

To help, think of it like this:

1. Go through each index of an array (i @ 0)
2. Go through each index of an array again (j @ 0)
3. If i isn't j and arr[i] is arr[j], then return both as an array.

```
function findPair(array) {
    for(var i = 0; i < array.length; i++) {
        var elemI = array[i];
        for(var j = 0; j < array.length; j++) {
            // Skip if "i" is "j"
            if (i == j) {
                continue;
            }
            var elemJ = array[j];
            if(elemI === elemJ) {
                // These two elements match, return them.
                return [elemI, elemJ];
            }
        }
    }
    // No elements were found!
    return null;
}
```

Flawless Pairs Solution

```
// Collect 4 pairs of gems.
// Each pair must contain equal valued gems.

// This function returns two items with the same value.
function findValuePair(items){
    // Check each possible pair in the array.
    // Iterate indexes 'i' from 0 to the last one.
    for (var i = 0; i < items.length; i++) {
        var itemI = items[i];
        // Iterate indexes 'j' from 0 to the last.
        for (var j = 0; j < items.length; j++) {
            // If it's the same element, then skip it.
            if (i == j) {
                continue;
            }
            var itemJ = items[j];
            // If we found a pair with two equal gems, then return them.
            if (itemI.value === itemJ.value) {
                return [itemI, itemJ];
            }
        }
    }
    // Return an empty array if no pair exists.
    return null;
}

while (true) {
    var gems = hero.findItems();
    var gemPair = findValuePair(gems);
    // If the gemPair exists, collect the gems!
    if (gemPair) {
        var gemA = gemPair[0];
        var gemB = gemPair[1];
        // Move to the first gem.
        hero.moveXY(gemA.pos.x, gemA.pos.y);
        // Return to get the haste from the wizard.
        hero.moveXY(40, 44);
        // Then move to the second gem.
        hero.moveXY(gemB.pos.x, gemB.pos.y);
        // Return to get the haste from the wizard.
        hero.moveXY(40, 44);
    }
}
```

#18. Twins Power

Level Overview and Solutions

Intro

We must lure this huge ogre through the temple of the Moon and the Sun. He is near-sighted and that will give us plenty of time to prepare a trap.

Use pairs of twins to awake the powers of the Sun and Moon!

Say each twin's name seperated by a space: `hero.say(twin1.id + " " + twin2.id)`

Default Code

```
// There are four pairs of twins, they should pray by pairs.
// You need to find twins and call them.

// Twins have the same names, only the last letter is different.
// This function checks if the pair of units are twins.
function areTwins(unit1, unit2) {
    var name1 = unit1.id;
    var name2 = unit2.id;
    if (name1.length !== name2.length) {
        return false;
    }
    for (var i = 0; i < name1.length - 1; i++) {
        if (name1[i] !== name2[i]) {
            return false;
        }
    }
    return true;
}

// Iterate over all pairs of paladins and
//  say() their name by pairs if they are twins.

// For example: hero.say("NameTwin1 NameTwin2")

// When twins are in their spots, lure the ogre.
// Don't be afraid of beams - they are dangerous only for ogres.
```

Overview

The code to check if two paladins are twins has been given to you: `areTwins(unit1, unit2)`.

You will need to iterate over each element each time you iternate over all the elements! Confusing, yes, but observe the following code for help:

```
for(var i = 0; i < array.length; i++) {
    var elemI = array[i];
    for(var j = 0; array.length; j++) {
        if (i == j) {
            continue;
        }
        var elemJ = array[j];
        // Check if the two elements match

            // Say each of the element's id.

    }
}
```

Be sure to move forward/backwards to lure the ogre through the trap.

Twins Power Solution

```
// There are four pairs of twins, they should pray by pairs.
// You need to find twins and call them.

// Twins have the same names, only the last letter is different.
// This function checks if the pair of units are twins.
function areTwins(unit1, unit2) {
    var name1 = unit1.id;
    var name2 = unit2.id;
    if (name1.length !== name2.length) {
        return false;
    }
    for (var i = 0; i < name1.length - 1; i++) {
        if (name1[i] !== name2[i]) {
            return false;
        }
    }
    return true;
}

// Iterate over all pairs of paladins and
//  say() their name by pairs if they are twins.
var friends = hero.findFriends();
for (var f1 = 0; f1 < friends.length; f1++) {
    for (var f2 = 0; f2 < friends.length; f2++) {
        if (f1 != f2 && areTwins(friends[f1], friends[f2])) {
            hero.say(friends[f1].id + " " + friends[f2].id);
        }
    }
}

// When twins are in their spots, lure the ogre.
// Don't be afraid of beams - they are dangerous only for ogres.
hero.moveXY(64, 36);
hero.moveXY(14, 36);
```

#19. Think Ahead

Level Overview and Solutions

Intro

Ogres have brought in a huge artillery cannon!

The soldiers are frantically avoiding being targeted, but it is no use! The huge cannon targets two units that are closest to each other.

Use this to your advantage and guide the wizards to apply Stoneskin to those two soldiers.

Default Code

```
// You need to distract "Big Bertha" until you special squad arrives.
// The cannon shoots at the pair of soldiers closest to each other.

// This function finds the pair of units
// with the minimum distance between them.
function findNearestPair(units) {
    var minDistance = 9001;
    var nearestPair = ["Nobody", "Nobody"];
    // You need to check and compare all pairs of units.
    // Iterate all units with indexes "i" from 0 to "units.length".

        // Iterate all units again with indexes "j".

            // If "i" is equal to "j", then skip (continue).

            // Find the distance between the i-th and j-th units.

            // If the distance less than 'minDistance':

                // Reassign 'minDistance' with the new distance.

                // Reassign 'nearestPair' to the names
                // of the current pair of units.

    return nearestPair;
}

while (true) {
    var soldiers = hero.findByType("soldier");
    // We know when the cannon shoots.
    if (hero.time % 8 === 5) {
        // Find which pair of soldiers in danger and protect them.
        var pairOfNames = findNearestPair(soldiers);
        // Say the soldier's names and wizards will protect them.
        hero.say(pairOfNames[0] + " " + pairOfNames[1]);
    }
}
```

Overview

The artillery will try to hit the two nearest units with each shot! Use this to your advantage and have your healers apply Stoneskin to the two closest soldiers.

You will need to iterate over all of our units and find the two whose distance is the smallest.

```
// The following code finds the two FURTHEST units
// In the level you need to find the two CLOSEST units

var maxDistance = 0;
var furthestPair = [null, null];
for(var i = 0; i < array.length; i++) {
    var itemI = array[i];
    for(var j = 0; j < array.length; j++) {
        // IMPORTANT: don't compare the same unit with itself
        if (i == j) {
            continue;
        }
        var itemJ = array[j];
        var dist = itemI.distanceTo(itemJ);
        // Check if the distance is greater than maxDistance

            // Create an array of the two elements and assign it to furthestPair

    }
}
```

Think Ahead Solution

```
// You need to distract "Big Bertha" until you special squad arrives.
// The cannon shoots at the pair of soldiers closest to each other.

// This function finds the pair of units
// with the minimum distance between them.
function findNearestPair(units) {
    var minDistance = 9001;
    var nearestPair = ["Nobody", "Nobody"];
    // You need to check and compare all pairs of units.
    // Iterate all units with indexes "i" from 0 to "units.length".
    for (var i = 0; i < units.length; i++) {
        // Iterate all units again with indexes "j".
        for (var j = 0; j < units.length; j++) {
            // If "i" is equal to "j", then skip (continue).
            if (i == j) {
                continue;
            }
            // Find the distance between the i-th and j-th units.
            var distance = units[i].distanceTo(units[j]);
            // If the distance less than 'minDistance':
            if (distance < minDistance) {
                // Reassign 'minDistance' with the new distance.
                minDistance = distance;
                // Reassign 'nearestPair' to the names
                // of the current pair of units.
                nearestPair = [units[i].id, units[j].id];
            }
        }
    }
    return nearestPair;
}

while (true) {
    var soldiers = hero.findByType("soldier");
    // We know when the cannon shoots.
    if (hero.time % 8 === 5) {
        // Find which pair of soldiers in danger and protect them.
        var pairOfNames = findNearestPair(soldiers);
        // Say the soldier's names and wizards will protect them.
        hero.say(pairOfNames[0] + " " + pairOfNames[1]);
    }
}
```

#20. Grid Search

Level Overview and Solutions

Intro

It was written in an old treasure map:

"Between four trees, between four rocks. The treasure is at the point `?0, ?0.` "

But the numbers are unreadable!

`?0` could mean 10, 20, 30... or more!

We found the trees and the rocks, so you need just look over each of the possible points.

Default Code

```
// The treasure somewhere between trees.
// The coordinates are 'x: ?0, y: ?0'.
// Move at all potential points and show to peasants where to dig.

var leftBorder = 20;
var rightBorder = 61;
var bottomBorder = 20;
var topBorder = 51;
var step = 10;

// Iterate X coordinates from the left to right border with step 10.
for (var x = leftBorder; x < rightBorder; x += step) {
    // Use a nested loop to iterate through Y coordinates for each X.
    // Iterate y coordinates from the bottom to top border with step 10.

        // Move to the point with coordinates X, Y and say anything.

}

// Allow peasants to check the last point.
hero.moveXY(20, 10);
```

Overview

Navigate across the field to find the treasure. You have 2 peasants at your disposal. Command them using `hero.say` to dig up at your location.

Note how the nested `for-loop` s map out the locations you should explore!

```
var startX = 20;
var endX = 110;
var startY = 30;
var endY = 90;
for(var x = startX; x < endX; x += 10) {
    for(var y = startY; y < endY; y += 10) {
        hero.moveXY(x, y);
            // The hero will move to each tile in a 100x100 square:
                // 10, 10 -> 90, 10
                //   |   ...   |
                //   V   ...   V
                // 90, 10 -> 90, 90
    }
}
```

Grid Search Solution

```
// The treasure somewhere between trees.
// The coordinates are 'x: ?0, y: ?0'.
// Move at all potential points and show to peasants where to dig.

var leftBorder = 20;
var rightBorder = 61;
var bottomBorder = 20;
var topBorder = 51;
var step = 10;

// Iterate X coordinates from the left to right border with step 10.
for (var x = leftBorder; x < rightBorder; x += step) {
    // Use a nested loop to iterate through Y coordinates for each X.
    // Iterate y coordinates from the bottom to top border with step 10.
    for (var y = bottomBorder; y < topBorder; y += step) {
        // Move to the point with coordinates X, Y and say anything.
        hero.moveXY(x, y);
        hero.say("Try here!");
    }
}

// Allow peasants to check the last point.
hero.moveXY(20, 10);
```

#21. Grid Minefield

Level Overview and Solutions

Intro

An ogre formation is marching towards the village. We much prepare the meeting for our "guests". We'll use their strict formation against them!

If we place mines between the ogre's lines and blow them all at once, the ogres will go flying.

Use nested loops to build the grid minefield, but, increment by numbers higher than one and use those as coordinates to place mines!

These mines won't explode unless a human steps on them, so don't be afraid and take a step!

Default Code

```
// The ogre formation is marching at the village.
// We have 90 seconds to build a minefield.
// We'll use their strict formation against them.

// Use nested loops to build the grid minefield.
// First iterate x coordinates from 12 to 60 with step 8.
for (var x = 12; x < 12 + 8 * 6; x += 8) {
    // For each x iterate y cordinates from 12 to 68 with step 8.
    for (var y = 12; y < 12 + 8 * 7; y += 8) {
        // For each point build "fire-trap" there.

    }
    // After each column, it's better to move right to avoid own traps.
    hero.moveXY(hero.pos.x + 8, hero.pos.y);
}

// Now wait and watch the coming ogres.
// When they are near (about 20 metres from the hero) blow mines with your hero.
// Just move at the nearest mine when it's the time.
```

Overview

It is possible to use `for-loop` s to map out not just 1-dimensional positions!

Nested `for-loop` s will cover each individual square, great for building an impenetrable minefield!

```
for(var x = 10; x < 100; x += 10) {
    for(var y = 10; y < 100; y += 10) {
        hero.moveXY(x, y);
            // The hero will move to each tile in a 100x100 square:
                // 10, 10 -> 90, 10
                //   |   ...   |
                //   V   ...   V
                // 90, 10 -> 90, 90
    }
}
```

Grid Minefield Solution

```
// The ogre formation is marching at the village.
// We have 90 seconds to build a minefield.
// We'll use their strict formation against them.

// Use nested loops to build the grid minefield.
// First iterate x coordinates from 12 to 60 with step 8.
for (var x = 12; x < 12 + 8 * 6; x += 8) {
    // For each x iterate y cordinates from 12 to 68 with step 8.
    for (var y = 12; y < 12 + 8 * 7; y += 8) {
        // For each point build "fire-trap" there.
        hero.buildXY("fire-trap", x, y);
    }
    // After each column, it's better to move right to avoid own traps.
    hero.moveXY(hero.pos.x + 8, hero.pos.y);
}

// Now wait and watch the coming ogres.
// <= near_blow %>
// Just move at the nearest mine when it's the time.
while (true) {
    var nearest = hero.findNearest(hero.findEnemies());
    if (nearest && hero.distanceTo(nearest) < 20) {
        hero.moveXY(52, 53);
        break;
    }
}
```

#22. To Arms!

Level Overview and Solutions

Intro

Ogres are coming to attack, but your army is asleep!

Use nested `for-loop` s to move through the camp row by row.

At each X-mark, stop and wake your troops with a `hero.say()` !

Default Code

```
// Ogres are going to attack soon.
// Move near each of tents (to the X marks)
// say() something at each X to wake your soldiers.
// Beware: leave the camp when the battle begins!
// Ogres will send reinforcements if they see the hero.

// The sergeant knows the distance between tents.
var sergeant =  hero.findNearest(hero.findFriends());

// The distances between the X marks.
var stepX = sergeant.tentDistanceX;
var stepY = sergeant.tentDistanceY;
// The number of tents.
var tentsInRow = 5;
var tentsInColumn = 4;

// The first tent mark has constant coordinates.
var firstX = 10;
var firstY = 14;

// Use nested loops and visit all 20 tents.
// IMPORTANT: move row by row - it's faster.

        // Move at the marks near tents and say anything.


// Now watch the battle.
```

Overview

Ogres are coming to attack, and your army is asleep! Wake them up by visiting each tent one-by-one.

Move to the X Mark in front of each tent and use `hero.say()` to wake the soldiers up.

There are 4 rows with 5 tents in each row.

The sergeant knows the exact distance between tents - every camp is a little different, so don't hard-code the values!

Find the camp's distance usin the sergeant's `tentDistanceX` and `tentDistanceY` .

```
var sergeant = hero.findNearest(hero.findFriends());
var stepX = sergeant.textDistanceX;
var stepY = sergeant.tentDistanceY;
```

Using this information, construct two `for-loop` s to navigate the camp:

```
var maxY = firstY + tentsInColumn * stepY;
var maxX = firstX + tentsInRow * stepX;

for(var y = firstY; y < maxY; y += stepY) {
    for(var x = firstX; x < maxX; x += stepX) {
        // Now move and say!

    }
}
```

Tip: It is quicker to do **row-by-row** instead of column-by-column.

To Arms! Solution

```
// Ogres are going to attack soon.
// Move near each of tents (to the X marks)
// say() something at each X to wake your soldiers.
// Beware: leave the camp when the battle begins!
// Ogres will send reinforcements if they see the hero.

// The sergeant knows the distance between tents.
var sergeant =  hero.findNearest(hero.findFriends());

// The distances between the X marks.
var stepX = sergeant.tentDistanceX;
var stepY = sergeant.tentDistanceY;
// The number of tents.
var tentsInRow = 5;
var tentsInColumn = 4;

// The first tent mark has constant coordinates.
var firstX = 10;
var firstY = 14;

// Use nested loops and visit all 20 tents.
// IMPORTANT: move row by row - it's faster.
for (var y = firstY; y < firstY + stepY * 4; y += stepY) {
    for (var x = firstX; x < firstX + stepX * 5; x += stepX){
        // Move at the marks near tents and say anything.
        hero.moveXY(x, y);
        hero.say("Wake UP!");
    }
}

// Now watch the battle.
```

#23. Power Points

Level Overview and Solutions

Intro

To pass the trial you must defeat 3 dark creatures. But first, you need to find them. To summon a skeleton or something useful you should stay on a point of the Power and *say "VENI"*.

But there are too many possible points and if the spell is said in the wrong place, then it can be bad (for you). Only wizards can see the points of the Power. Luckily you know one of them and he prepared the map for you.

The map is represented as a 2-dimensional array (grid) of numbers. `0` is a wrong place. Positive numbers are power points. Iterate through the grid, find the points and meet your challenges.

Default Code

```
// You need to find and destroy 3 skeletons.
// Skeletons and items are summoned at points of power.
// Move to a point and say the spell: "VENI".
// To find the required points, use the wizard's map.
// 0s are bad points. Positive numbers are good.

var spell = "VENI";
// The map of points is a 2D array of numbers.
var wizard = hero.findNearest(hero.findFriends());
var powerMap = wizard.powerMap;

// This function converts grid into x-y coordinates.
function convert(row, col) {
    return {x: 16 + col * 12, y: 16 + row * 12};
}

// Loop through the powerMap to find positive numbers.
// First, loop through indexes of rows.
for (var i = 0; i < powerMap.length; i++) {
    // Each row is an array. Iterate through it.
    for (var j = 0; j < powerMap[i].length; j++) {
        // Get the value of the i row and j column.
        var pointValue = powerMap[i][j];
        // If it's a positive number:

            // Use convert to get XY coordinates.

            // Move there, say "VENI" and be prepared!

    }
}
```

Overview

The map that we get from the wizard is a 2-dimensional array of numbers. It's an array where each element is an array of number. It looks like this:

```
var powerMap = [
    [0, 1, 0, 0, 0],
    [2, 0, 0, 3, 0],
    [0, 0, 4, 0, 0],
    [0, 0, 0, 0, 5]];
```

To iterate all elements from that arrays we need to use two concepts from the previous levels: nested loops and 2d array elements reading.

First we should iterate all elements of the root array which are arrays too:

```
for (var i = 0; i < powerMap.length; i++) {
    ...
}
```

Next, use the nested loop to iterate through each array. You can save get a row inside the loop definition or save it in a variable for the each inner loop. As the result we have values of the row and the column for all elements and can get their value:

```
for (var i = 0; i < powerMap.length; i++) {
    for (var j = 0; j < powerMap[i]; j++) {
        var value = powerMap[i][j];
    }
}
```

Power Points Solution

```
// You need to find and destroy 3 skeletons.
// Skeletons and items are summoned at points of power.
// Move to a point and say the spell: "VENI".
// To find the required points, use the wizard's map.
// 0s are bad points. Positive numbers are good.

var spell = "VENI";
// The map of points is a 2D array of numbers.
var wizard = hero.findNearest(hero.findFriends());
var powerMap = wizard.powerMap;

// This function converts grid into x-y coordinates.
function convert(row, col) {
    return {x: 16 + col * 12, y: 16 + row * 12};
}

// Loop through the powerMap to find positive numbers.
// First, loop through indexes of rows.
for (var i = 0; i < powerMap.length; i++) {
    // Each row is an array. Iterate through it.
    for (var j = 0; j < powerMap[i].length; j++) {
        // Get the value of the i row and j column.
        var pointValue = powerMap[i][j];
        // If it's a positive number:
        if (powerMap[i][j] > 0) {
            // Use convert to get XY coordinates.
            var coor = convert(i, j);
            // Move there, say "VENI" and be prepared!
            hero.moveXY(coor.x, coor.y);
            hero.say("veni");
            var enemy = hero.findNearest(hero.findEnemies());
            while (enemy && enemy.health > 0) {
                hero.attack(enemy);
            }
            var item = hero.findNearest(hero.findItems());
            if (item) {
                hero.moveXY(item.pos.x, item.pos.y);
            }
        }
    }
}
```

#24. Danger Valley

Level Overview and Solutions

Intro

Set up a field of mines to trap incoming ogres!

The answer lies in a set of nested arrays, so lay out the fire-traps carefully.

Default Code

```
// Ogres have taken some peasants hostage!
// The scouts have given you intel for an ambush.
// this.grid holds an array of arrays.
// Inside the sub-arrays, 0 is a peasant, 1 is an ogre.
// Use this information to setup fire-traps.

// Remember the containing array is just an array!
// Iterate over all the elements of this array.
for(var i = 0; i < hero.grid.length; i++) {
    var row = hero.grid[i];
    // Now, row is just another array!
    // Iterate over all the tiles in this array:

        // Check if the tile at i, j is 1 to build:
        //hero.buildXY("fire-trap", 36 + 6 * j, 20 + 6 * i);

}
// Finally, retreat back to cover.
```

Overview

Ogres have captured a group of peasants, and it's your job to save them.

Luckily a scout has gone ahead and found out what the layout of the ogres will be and reported this information back as nested arrays.

A nested array, or 2D array, is an array containing... more arrays! Look at the following code:

```
var doubleArray = [[1, 2],[3, 4]];
```

A bit confusing at first, but if we clean up the code, we can better understand how it works:

```
var doubleArray = [
    [1,2],
    [3,4]
];
```

Now that the structure is more apparent, to access a specific element you will use the familiar `[index]` notation:

```
var doubleArray = [
    [1,2],
    [3,4]
];
var row1 = doubleArray[0]; // This is [1,2]!
var cell1 = row1[0]; // This is 1!
```

However, if you're real crafty, you can double up these index calls like so:

```
var doubleArray = [
    [1,2],
    [3,4]
]
var cell1 = doubleArray[0][0]; // This is 1!
```

Danger Valley Solution

```
// Ogres have taken some peasants hostage!
// The scouts have given you intel for an ambush.
// this.grid holds an array of arrays.
// Inside the sub-arrays, 0 is a peasant, 1 is an ogre.
// Use this information to setup fire-traps.

// Remember the containing array is just an array!
// Iterate over all the elements of this array.
for(var i = 0; i < hero.grid.length; i++) {
    var row = hero.grid[i];
    // Now, row is just another array!
    // Iterate over all the tiles in this array:
    for(var j = 0; j < row.length; j++) {
        // Check if the tile at i, j is 1 to build:
        var tile = row[j];
        if(tile === 1) {
            hero.buildXY("fire-trap", 36 + 6 * j, 20 + 6 * i);
        }
    }
}
// Finally, retreat back to cover.
hero.moveXY(29, 55);
```

#25. Sleepwalkers

Level Overview and Solutions

Intro

This village is an anomaly. There are too many sleepwalkers and they often walk out the village. But it's only half of the problem. There are sleepwalking yetis and they are following our returning peasants.

We must build a fence system to pass sleeping peasants and turn round yetis. Don't use combat methods which can wake up yetis. Yetis are angry when somebody wakes them up.

Senick is an experienced hunter and he knows everything about yetis. He can calculate and predict their routes. Senick prepared the grid map of the fence system. You should use that scheme and build fences where it's required.

The map is represented as 2d-array (an array of arrays) with numbers (1 or 0). 0 is an empty cell, 1 is a place where to build a fence.

Default Code

```
// Our sleepwalking peasants are returning.
// But sleeping yetis are also coming.
// DON'T WAKE THEM UP!
// Build fences to let peasants through and stop yetis.


// Senick's prepared the grid map how to build fences.
var hunter = hero.findNearest(hero.findFriends());
var fenceMap = hunter.getMap();

// This function converts grid into XY coordinates.
function convertCoor(row, col) {
    return {x: 34 + col * 4, y: 26 + row * 4};
}


// Iterate over fenceMap and build at fence at all 1s.


// Move back to the village after building the fences.
```

Overview

The grid map is an array of arrays where each element can be 1 or 0. 0 is an empty cell, 1 is a place where to build a fence.

In previous levels you've learnt how to loop through 2d-arrays and get their element values. Use nested loops and `array[i][j]` syntax to read all values and if you meet 1, then build a fence there. There is a prepared function `convertCoor` to convert grid coordinates (row and column) to (x, y) coordinates.

```
// 2-nd row, 3-rd column
convertCoor(2, 4); // result: {x: 42, y: 38}
```

Don't forget in the village when fences are built, because yetis can smell the hero.

Sleepwalkers Solution

```
// Our sleepwalking peasants are returning.
// But sleeping yetis are also coming.
// DON'T WAKE THEM UP!
// Build fences to let peasants through and stop yetis.


// Senick's prepared the grid map how to build fences.
var hunter = hero.findNearest(hero.findFriends());
var fenceMap = hunter.getMap();

// This function converts grid into XY coordinates.
function convertCoor(row, col) {
    return {x: 34 + col * 4, y: 26 + row * 4};
}


// Iterate over fenceMap and build at fence at all 1s.
for (var i = 0; i < fenceMap.length; i++) {
    for (var j = 0; j < fenceMap[i].length; j++) {
        if (fenceMap[i][j]) {
            var pos = convertCoor(i, j);
            hero.moveXY(pos.x, pos.y);
            hero.buildXY("fence", pos.x, pos.y);
        }
    }
}

// Move back to the village after building the fences.
hero.moveXY(29, 17);
```

#26. Cannon Landing Force

Level Overview and Solutions

Intro

Frontier colonists need our help and we can send some soldiers to the mountain village. Also they asked to clear the square from old fire traps. How will we do it? With cannons of course!

The artillery can launch soliers and anti-trap shell over mountains. Peasants meet us on the square, so be careful to launch soldiers.

We have a map as a 2 dimensional array (a grid), so you can get the values of the specific cells.

Default Code

```
// We should send soldiers to defend the village.
// Also we need clear out old fire traps.
// For both of those goals we'll use the artillery!
// The artillery can launch soldiers and anti-traps.

// The scout prepared a map of the landing zone.
// The map is a 2D array where cells are strings.
var landingMap = hero.findNearest(hero.findFriends()).landingMap;

// Tell the cannons the row, column, and target type.
// To get the element, use array[i][j]
// First, let's look at row 0 and column 0.
var cell = landingMap[0][0];
// Next, say the coordinates and what's there.
hero.say("row 0 column 0 " + cell);

// Next cell is the 3rd row and the 2nd column.
hero.say("row 3 column 2 " + landingMap[3][2]);

// Now do it yourself for the next point:
// The 2nd row and 1st column.

// The 1st row and 0th column.

// The 0th row and 2nd column.

// The 1st row and 3rd column.
```

Overview

The map of the square is represented as a 2-dimensional array (a grid) with string values. The 2-dimensional array is an array where each element is an array. As the result, we get something like a spreadsheet. Arrays inside can have various lengths, but for our case, they all have the same length. Usually, if you use 2d arrays to represent a spreadsheet or a map it's useful to use the same length for each row.

2d array can be written this way:

```
var grid = [[0, 1, 2],
            [3, 4, 5]];
```

In the above example the grid contains 2 rows and 3 columns. If we need to get the value of specific elements we can use the next syntax (don't forget the first row has the index 0):

```
// get the first element
grid[0][0]; // it's 0
// the last element
grid[1][2]; // 5
// the row 1 the column 0
grid[1][0]; // 3
```

The map of the town square on this level is represented as a 2d array of strings. Each cell contain the string value: "peasant", "trap" or "clear".

For example (the level grid can have another size):

```
var landingMap = [
    ["peasant", "clear", "clear", "trap"],
    ["trap", "clear", "peasant", "peasant"],
    ["clear", "trap", "clear", "clear"]];
```

Rows and columns are counted from zero in the directions as X and Y coordinates. To shoot the cannons need to know grid coordinates (the row and the column) and what in that cell is.

So if you need to shoot at the row 2 and the column 3, then first get the value at that cell and say this:

```
var cellValue = landingMap[2][3]; // clear
hero.say("Row 2 column 3 - " + cellValue);
```

If the cannons hear "trap" or "clear" in your commands, then one of them (depends on the type) shoots. Be careful and give only correct coordinates. For example, if the grid's size is 3x4

```
// Wrong coordinates for the cannon. BOOM!
hero.say("Row 9001 column 19 - " + landingMap[2][3]);
// Wrong indexes to access. ReferenceError.
hero.say("Row 2 column 3 - " + landingMap[9001][19]);
```

Use cell coordinates that are given in the sample code to be sure for the success.

Cannon Landing Force Solution

```
// We should send soldiers to defend the village.
// Also we need clear out old fire traps.
// For both of those goals we'll use the artillery!
// The artillery can launch soldiers and anti-traps.

// The scout prepared a map of the landing zone.
// The map is a 2D array where cells are strings.
var landingMap = hero.findNearest(hero.findFriends()).landingMap;

// Tell the cannons the row, column, and target type.
// To get the element, use array[i][j]
// First, let's look at row 0 and column 0.
var cell = landingMap[0][0];
// Next, say the coordinates and what's there.
hero.say("row 0 column 0 " + cell);

// Next cell is the 3rd row and the 2nd column.
hero.say("row 3 column 2 " + landingMap[3][2]);

// Now do it yourself for the next point:
// The 2nd row and 1st column.
hero.say("row 2 column 1 " + landingMap[2][1]);
// The 1st row and 0th column.
hero.say("row 1 column 0 " + landingMap[1][0]);
// The 0th row and 2nd column.
hero.say("row 0 column 2 " + landingMap[0][2]);
// The 1st row and 3rd column.
hero.say("row 1 column 3 " + landingMap[1][3]);
```

#27. Snowdrops

Level Overview and Solutions

Intro

Spring is coming! The time for the big cleaning! We need to clear the forest and we have the cannon and the forest map for that. But the map has rows with different lengths, so be sure before a shot. We can't afford to shot everywhere - ammo is expensive and we have just enough to clean traps.

The map is an array of arrays, where 1 is clear and 0 is a trap. Like:

```
[[1, 1, 0],
 [0, 1],
 [1, 1, 0, 0, 1]]
```

Say the row and the column of the map where the artillery should shoot. We prepared the cell coordinates (you'll find them in the sample code) which you need to check.

Default Code

```
// We need to clear the forest of traps!
// The scout prepared a map of the forest.
// But be careful where you shoot! Don't start a fire.

// Get the map of the forest.
var forestMap = hero.findNearest(hero.findFriends()).forestMap;

// The map is a 2D array where 0 is a trap.
// The first sure shot.
hero.say("Row " + 0 + " Column " + 1 + " Fire!");

// But for the next points, check before shooting.
// There are an array of points to check.
var cells = [{row: 0, col: 4}, {row: 1, col: 0}, {row: 1, col: 2}, {row: 1, col: 4},
    {row: 2, col: 1}, {row: 2, col: 3}, {row: 2, col: 5}, {row: 3, col: 0},
    {row: 3, col: 2}, {row: 3, col: 4}, {row: 4, col: 1}, {row: 4, col: 2},
    {row: 4, col: 3}, {row: 5, col: 0}, {row: 5, col: 3}, {row: 5, col: 5},
    {row: 6, col: 1}, {row: 6, col: 3}, {row: 6, col: 4}, {row: 7, col: 0}];

for (var i = 0; i < cells.length; i++) {
    var row = cells[i].row;
    var col = cells[i].col;
    // If row is less than forestMap length:

        // If col is less than forestMap[row] length:

            // Now, we know the cell exists.
            // If it is 0, say where to shoot:

}
```

Overview

The map is an array of arrays, where 1 is clear and 0 is a trap. Like:

```
var forestMap = [
    [1, 1, 0],
    [0, 1],
    [1, 1, 0, 0, 1]]
```

As we can see rows of that arrays have different lengths. So we need to check the correctness of coordinates before to get access, else you can get an error.

```
// The correct row, but the wrong column:
forestMap[0][4] // undefined
// The wrong row, any column
forestMap[5][1] // TypeError, because forestMap[5] is undefined
```

It's a good practice to check indexes before access them if you aren't sure it. For example, if you get access to random cell by random or outer source coordinates.

To check the existence of the cell you can check if it's less than the length of the array (We don't consider negative indexes here):

```
var row = 1;
var col = 3;
if (row < forestMap.length) {
    // if 'row' is correct index
    if (col < forestMap[row].length) {
        // now we can get the cell
        hero.say(forestMap[row][col]);
    }
}
```

It can be written shorter with logic operators:

```
// The second part is checked only if the first is true
if (row < forestMap.length && col < forestMap[row].length) {
        hero.say(forestMap[row][col]);
}
```

Snowdrops Solution

```
// We need to clear the forest of traps!
// The scout prepared a map of the forest.
// But be careful where you shoot! Don't start a fire.

// Get the map of the forest.
var forestMap = hero.findNearest(hero.findFriends()).forestMap;

// The map is a 2D array where 0 is a trap.
// The first sure shot.
hero.say("Row " + 0 + " Column " + 1 + " Fire!");

// But for the next points, check before shooting.
// There are an array of points to check.
var cells = [{row: 0, col: 4}, {row: 1, col: 0}, {row: 1, col: 2}, {row: 1, col: 4},
    {row: 2, col: 1}, {row: 2, col: 3}, {row: 2, col: 5}, {row: 3, col: 0},
    {row: 3, col: 2}, {row: 3, col: 4}, {row: 4, col: 1}, {row: 4, col: 2},
    {row: 4, col: 3}, {row: 5, col: 0}, {row: 5, col: 3}, {row: 5, col: 5},
    {row: 6, col: 1}, {row: 6, col: 3}, {row: 6, col: 4}, {row: 7, col: 0}];

for (var i = 0; i < cells.length; i++) {
    var row = cells[i].row;
    var col = cells[i].col;
    // If row is less than forestMap length:
    if (row < forestMap.length) {
        // If col is less than forestMap[row] length:
        if (col < forestMap[row].length) {
            // Now, we know the cell exists.
            // If it is 0, say where to shoot:
            if (forestMap[row][col] === 0) {
                hero.say("Row " + row + " Column " + col + " Fire!");
            }
        }
    }
}
```

#28. Reindeer Wakeup

Level Overview and Solutions

Intro

It's time for the reindeer to get up and start their day! Merek the old herder is the only one they'll listen to, but he's blind and needs help to tell who's awake and who's asleep.

Use an *array* to keep track of which reindeer are awake and in the field, and which are asleep in their pens. Tell Merek which reindeer are awake, and then he'll tell them to get up and play.

Default Code

```
// This array contains the status for each reindeer.
var deerStatus = [ "asleep", "asleep", "asleep", "asleep", "asleep" ];

// And this array contains our reindeer.
var friends = hero.findFriends();

// Loop through the reindeer and find the awake ones:
for (var deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    var reindeer = friends[deerIndex];

    // Reindeer with y position > 30 aren't in a pen.
    // If so, set the reindeer's entry to "awake".

}

// Loop through statuses and report to Merek.
for (var statusIndex = 0; statusIndex < deerStatus.length; statusIndex++) {
    // Tell Merek the reindeer index and its status.
    // Say something like "Reindeer 2 is asleep".

}
```

Overview

In this level, you will use *arrays* to keep track of the reindeer. You start with two arrays:

1. An array that is used to mark which reindeer are asleep and which are awake;
2. An array of the reindeer themselves.

The first step is to figure out which reindeer are awake. To do this, loop through the reindeer. For each one, check its position. If its Y coordinate is more than 30, then it's not in its pen and it must be awake. When a reindeer is awake, mark its slot in the `deerStatus` array to "awake".

After figuring out which reindeer are up, loop through the `deerStatus` array and tell Merek what's what. If he hears you say that a reindeer is "asleep", he'll command it to wake up.

If everything goes well, all the reindeer will be awake and ready to start the day!

By now, you're used to looping over arrays using `while` and `for` loops:

```
for (var i = 0; i < enemies.length; i++) {
    var enemy = enemies[i];
}
```

But you can access any element of an array at any time in any order, as long as it exists:

```
var a = [ null, "one", "two" ];
hero.say(a[1]); // Says "one"
hero.say(a[2]); // Says "two"
hero.say(a[0]); // Says nothing
hero.say(a[3]); // Error! Array starts at 0 and ends at 2.
```

You can also change the elements of an array however you like:

```
var a = [ null, "no", "maybe" ];
hero.say(a[1]); // Says "no"
a[1] = "yes"
hero.say(a[1]); // Says "yes"
```

You'll have to make some changes to the `deerStatus` array to solve this level!

Reindeer Wakeup Solution

```
// This array contains the status for each reindeer.
var deerStatus = [ "asleep", "asleep", "asleep", "asleep", "asleep" ];

// And this array contains our reindeer.
var friends = hero.findFriends();

// Loop through the reindeer and find the awake ones:
for (var deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    var reindeer = friends[deerIndex];

    // Reindeer with y position > 30 aren't in a pen.
    // If so, set the reindeer's entry to "awake".
    if (reindeer.pos.y > 30) {
        deerStatus[deerIndex] = "awake";
    }
}

// Loop through statuses and report to Merek.
for (var statusIndex = 0; statusIndex < deerStatus.length; statusIndex++) {
    // Tell Merek the reindeer index and its status.
    // Say something like "Reindeer 2 is asleep".
    hero.say("Reindeer " + statusIndex + " is " + deerStatus[statusIndex]);
}
```

#29. Reindeer Spotter

Level Overview and Solutions

Intro

Merek the blind reindeer herder wants to put the reindeer down for the night, but he can't tell which pens already have sleeping reindeer in them!

Use an *array* to keep track of which pens have reindeer in them and which are free, then tell Merek which spots are open. He'll tell the deer where to go, and then everyone can get some rest.

Default Code

```
// This array contains each of the pen's positions.
var penPositions = [ {"x":20,"y":24}, {"x":28,"y":24}, {"x":36,"y":24}, {"x":44,"y":24}, {"x":52,"y":24}
    ];

// This array is used to track which reindeer is in it.
var penOccupants = [ "empty", "empty", "empty", "empty", "empty" ];

// And this array contains our reindeer.
var friends = hero.findFriends();

// Figure out which reindeer are already in their pens.
for (var deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    var reindeer = friends[deerIndex];

    // Check each pen if it matches a reindeer's pos.
    for (var penIndex = 0; penIndex < penPositions.length; penIndex++) {
        var penPos = penPositions[penIndex];

        if (penPos.x == reindeer.pos.x && penPos.y == reindeer.pos.y) {
            // Put the id in penOccupants at penIndex.

            // break out of the inner loop here.

        }
    }
}

// Tell Merek what's in each pen.
for (var occIndex = 0; occIndex < penOccupants.length; occIndex++) {
    // Tell Merek the pen index and the occupant.
    // Say something like "Pen 3 is Dasher"

}
```

Overview

In this level, you will use *arrays* to keep track of the reindeer. You start with three arrays:

1. An array of positions of the pens that the reindeer will go into.
2. An array that is used to mark which pens have reindeer in them.
3. An array of the reindeer themselves.

The first step is to figure out which reindeer are already in their pens. To do this, loop through the reindeer. For each one, loop through each pen position. If the pen position matches the reindeer's position, it's already there. Use the `penOccupants` array to keep track of which pens have reindeer in them.

After figuring out which pens are occupied, loop through the `penOccupants` array and tell Merek what's in each one. If he hears you say that a pen is "empty," he'll command a reindeer to move into that pen.

If everything goes well, you'll have one reindeer sleeping peacefully in each pen!

By now, you're used to looping over arrays using `while` and `for` loops:

```
for (var i = 0; i < enemies.length; i++) {
    var enemy = enemies[i];
}
```

But you can access any element of an array at any time in any order, as long as it exists:

```
var a = [ null, "one", "two" ];
hero.say(a[1]); // Says "one"
hero.say(a[2]); // Says "two"
hero.say(a[0]); // Says nothing
hero.say(a[3]); // Error! Array starts at 0 and ends at 2.
```

You can also change the elements of an array however you like:

```
var a = [ null, "no", "maybe" ];
hero.say(a[1]); // Says "no"
a[1] = "yes"
hero.say(a[1]); // Says "yes"
```

You'll need to make a number of changes to the `penOccupants` array to get through this level!

Reindeer Spotter Solution

```
// This array contains each of the pen's positions.
var penPositions = [ {"x":20,"y":24}, {"x":28,"y":24}, {"x":36,"y":24}, {"x":44,"y":24}, {"x":52,"y":24}
    ];

// This array is used to track which reindeer is in it.
var penOccupants = [ "empty", "empty", "empty", "empty", "empty" ];

// And this array contains our reindeer.
var friends = hero.findFriends();

// Figure out which reindeer are already in their pens.
for (var deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    var reindeer = friends[deerIndex];

    // Check each pen if it matches a reindeer's pos.
    for (var penIndex = 0; penIndex < penPositions.length; penIndex++) {
        var penPos = penPositions[penIndex];

        if (penPos.x == reindeer.pos.x && penPos.y == reindeer.pos.y) {
            // Put the id in penOccupants at penIndex.
            penOccupants[penIndex] = reindeer.id;
            // break out of the inner loop here.
            break;
        }
    }
}

// Tell Merek what's in each pen.
for (var occIndex = 0; occIndex < penOccupants.length; occIndex++) {
    // Tell Merek the pen index and the occupant.
    // Say something like "Pen 3 is Dasher"
    hero.say("Pen " + occIndex + " is " + penOccupants[occIndex]);
}
```

#30. Reindeer Tender

Level Overview and Solutions

Intro

These reindeer have had a long day of playing in the snow, eating hay, and whatever else it is that reindeer do. Now you need to put them in their pens so they can get some sleep.

Use arrays to keep track of which pens already have reindeer in them, and assign the other reindeer to empty stalls.

Default Code

```
// This is the array of pen positions
var penPositions = [ {"x":20,"y":24}, {"x":28,"y":24}, {"x":36,"y":24}, {"x":44,"y":24}, {"x":52,"y":24}
    ];

// Use this array to keep track of each pen's reindeer.
var penOccupants = [ null, null, null, null, null ];

// And this array contains our reindeer.
var friends = hero.findFriends();

// Figure out which reindeer are already in their pens.
for (var deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    var reindeer = friends[deerIndex];

    // For each position check if it matches a reindeer.
    for (var penIndex = 0; penIndex < penPositions.length; penIndex++) {
        var penPos = penPositions[penIndex];

        if (penPos.x == reindeer.pos.x && penPos.y == reindeer.pos.y) {
            // Put the reindeer in occupants at penIndex

            // Remove the reindeer from the friends array.

            // break out of the inner loop here:

        }
    }
}

// Assign the remaining reindeer to new positions.
for (deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    // If the reindeer is null, use continue:


    // Look for the first pen with nothing.
    for (var occIndex = 0; occIndex < penOccupants.length; occIndex++) {
        // If there is nothing, the pen is open:
        if (penOccupants[occIndex] === null) {
            // Put the reindeer in the occupants array.

            // Command the reindeer to move to the pen.

            // break out early so we don't reassign:

        }
    }
}
```

Overview

In this level, you will use *arrays* to organize the reindeer. You start with three arrays:

1. An array of positions that the reindeer will go to.
2. An array that is used to mark which reindeer are in which pens.
3. An array of the reindeer themselves.

The first step is to figure out which reindeer are already in their pens. To do this, loop through the reindeer. For each one, loop through each pen position. If the pen position matches the reindeer's position, it's already there. Use the `penOccupants` and `friends` arrays to keep track of which reindeer are already asleep -- when a reindeer is in a pen, move the reindeer from the `friends` array to the `penOccupants` array to track it.

Next, you'll go through the `friends` array again to look at the remaining reindeer (the ones that *aren't* already in pens). For each one, Look through the `penOccupants` array for the first spot that *doesn't* have a reindeer in it. When you find one, command the reindeer to move to the spot.

If everything goes well, you'll have one reindeer sleeping peacefully in each pen!

By now, you're used to looping over arrays using `while` and `for` loops:

```
for (var i = 0; i < enemies.length; i++) {
    var enemy = enemies[i];
}
```

But you can access any element of an array at any time in any order, as long as it exists:

```
var a = [ null, "one", "two" ];
hero.say(a[1]); // Says "one"
hero.say(a[2]); // Says "two"
hero.say(a[0]); // Says nothing
hero.say(a[3]); // Error! Array starts at 0 and ends at 2.
```

You can also change the elements of an array however you like:

```
var a = [ null, "no", "maybe" ];
hero.say(a[1]); // Says "no"
a[1] = "yes"
hero.say(a[1]); // Says "yes"
```

You'll need to change the `penOccupants` and `friends` arrays quite a bit to get through this level!

Reindeer Tender Solution

```
// This is the array of pen positions
var penPositions = [ {"x":20,"y":24}, {"x":28,"y":24}, {"x":36,"y":24}, {"x":44,"y":24}, {"x":52,"y":24}
    ];

// Use this array to keep track of each pen's reindeer.
var penOccupants = [ null, null, null, null, null ];

// And this array contains our reindeer.
var friends = hero.findFriends();

// Figure out which reindeer are already in their pens.
for (var deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    var reindeer = friends[deerIndex];

    // For each position check if it matches a reindeer.
    for (var penIndex = 0; penIndex < penPositions.length; penIndex++) {
        var penPos = penPositions[penIndex];

        if (penPos.x == reindeer.pos.x && penPos.y == reindeer.pos.y) {
            // Put the reindeer in occupants at penIndex
            penOccupants[penIndex] = reindeer;
            // Remove the reindeer from the friends array.
            friends[deerIndex] = null;
            // break out of the inner loop here:
            break;
        }
    }
}

// Assign the remaining reindeer to new positions.
for (deerIndex = 0; deerIndex < friends.length; deerIndex++) {
    // If the reindeer is null, use continue:
    reindeer = friends[deerIndex];
    if (reindeer === null)
        continue;

    // Look for the first pen with nothing.
    for (var occIndex = 0; occIndex < penOccupants.length; occIndex++) {
        // If there is nothing, the pen is open:
        if (penOccupants[occIndex] === null) {
            // Put the reindeer in the occupants array.
            penOccupants[occIndex] = reindeer;
            // Command the reindeer to move to the pen.
            hero.command(reindeer, "move", penPositions[occIndex]);
            // break out early so we don't reassign:
            break;
        }
    }
}
```

#31. Ritual of Rectangling

Level Overview and Solutions

Intro

That ogre is huge! But we know the geometric ritual and can summon the Ancient Warrior to fight for us.

We need four paladins which should form a rectangle with a specific perimeter and a specific area. When the retangle is formed, say: "VENITE"!

The paladins will continuously move, so watch them and say the magic word when the rectangle is correct.

Computers have trouble handling numbers with any certainty, so, use an 'almost equal' comparison to verify the rectangle is approximately the correct size. The magic spell has a 3% margin of error, so use that to your advantage!

Default Code

```
// We must summon the Ancient Warrior for this ogre!
// Four paladins must form a rectangle.
// But the rectangle needs a specific area and perimeter
// Paladins will keep moving, say the spell when ready.
// It is hard to be precise, but almost equal is good.

// This function should compare valueA and B within 3%.
function almostEqual(valueA, valueB) {
    // Check if valueA is > 0.97 and < 1.03 of valueB.

    // As a default, just check equality.
    return valueA === valueB;
}

// This function should calculate the perimeter:
function perimeter(side1, side2) {
    // The perimeter is the sum of all four sides.

}

// This function should return the area:
function area(side1, side2) {
    // The area of a rectangle is the product of 2 sides

}


// Required summoning values for area and perimeter:
var requiredPerimeter = 104;
var requiredArea = 660;

// We will use this unit as a base for our calculations:
var base = hero.findNearest(hero.findFriends());

while (true) {
    var sideSN = base.distanceTo("Femae");
    var sideWE = base.distanceTo("Illumina");
    var currentPerimeter = perimeter(sideSN, sideWE);
    var currentArea = area(sideSN, sideWE);
    if (almostEqual(currentArea, requiredArea) && almostEqual(currentPerimeter, requiredPerimeter)) {
        hero.say("VENITE!");
        break;
    }
}
```

Overview

These four paladins can summon a powerful unit to defeat this giant ogre! They will move around and you must tell them when they are forming the required rectangle.

Check their area and perimeter constantly until they meet the requirements.

Area of a rectangle is found by multiplying the length of two adjacent sides:

```
function area(sideA, sideB) {
    // Return the product of the multiplication of sideA and sideB.
}
```

Perimeter is found by adding the length of every side:

```
function perimeter(sideA, sideB) {
    // A rectangle has four sides!
    // Return the sum of sideA x 2 and sideB x 2!
}
```

Since the paladins are moving around, it is possible they will skip over the exact moment their perimeter and area are your desired values. This is why we use a 'approximately equal' function:

```
function approxEqual(numberA, numberB) {
    var min = numberA * 0.97;
    var max = numberA * 1.03;
    // Return true if numberB is above min and below max

    // Otherwise return false
    return false;
}
```

Ritual of Rectangling Solution

```
// We must summon the Ancient Warrior for this ogre!
// Four paladins must form a rectangle.
// But the rectangle needs a specific area and perimeter
// Paladins will keep moving, say the spell when ready.
// It is hard to be precise, but almost equal is good.

// This function should compare valueA and B within 3%.
function almostEqual(valueA, valueB) {
    // Check if valueA is > 0.97 and < 1.03 of valueB.
    if(valueA > 0.97 * valueB && valueA < 1.03 * valueB) {
        return true;
    }
    // As a default, just check equality.
    return valueA === valueB;
}

// This function should calculate the perimeter:
function perimeter(side1, side2) {
    // The perimeter is the sum of all four sides.
    return side1 * 2 + side2 * 2;
}

// This function should return the area:
function area(side1, side2) {
    // The area of a rectangle is the product of 2 sides
    return side1 * side2;
}


// Required summoning values for area and perimeter:
var requiredPerimeter = 104;
var requiredArea = 660;

// We will use this unit as a base for our calculations:
var base = hero.findNearest(hero.findFriends());

while (true) {
    var sideSN = base.distanceTo("Femae");
    var sideWE = base.distanceTo("Illumina");
    var currentPerimeter = perimeter(sideSN, sideWE);
    var currentArea = area(sideSN, sideWE);
    if (almostEqual(currentArea, requiredArea) && almostEqual(currentPerimeter, requiredPerimeter)) {
        hero.say("VENITE!");
        break;
    }
}
```

#32. Square Shield

Level Overview and Solutions

Intro

There are a lot of wild yetis and they are very agressive! Four paladins and a prayer can make the divine shield. Form a square with the paladins in the vertices and enemies will not pass.

Two paladins are ready, find the places for the other two.

Default Code

```
// Incoming yeti attack!
// Use your paladins to form a square!
// Command Illumina and Vaelia to create a square!

function findByName(name, thangs) {
    for(var i = 0; i < thangs.length; i++) {
        var thang = thangs[i];
        if(thang.id == name) {
            return thang;
        }
    }
    return null;
}
var friends = hero.findFriends();
var celadia = findByName("Celadia", friends);
var dedalia = findByName("Dedalia", friends);
var sideLength = celadia.distanceTo(dedalia);

// First assign the remaining paladins to variables:

// Command both to move to the corners of the square.
// Remember squares have equal-length sides!
```

Overview

To survive, you must assemble the Paladins into a square formation!

Since the Paladins will be making the corners of a square, consider these properties of a square:

1. The corners of a square share an x or y coordinate with an adjacent corner.
2. A square has equilateral (same length) sides.

To solve this level, move Vaelia and Illumina a `sideLength` away from each corner, but make sure they share a similar corner position (`x`).

Square Shield Solution

```
// Incoming yeti attack!
// Use your paladins to form a square!
// Command Illumina and Vaelia to create a square!

function findByName(name, thangs) {
    for(var i = 0; i < thangs.length; i++) {
        var thang = thangs[i];
        if(thang.id == name) {
            return thang;
        }
    }
    return null;
}
var friends = hero.findFriends();
var celadia = findByName("Celadia", friends);
var dedalia = findByName("Dedalia", friends);
var sideLength = celadia.distanceTo(dedalia);

// First assign the remaining paladins to variables:
var vaelia = findByName("Vaelia", friends);
var illumina = findByName("Illumina", friends);

// Command both to move to the corners of the square.
// Remember squares have equal-length sides!
hero.command(vaelia, "move", {x: celadia.pos.x, y: celadia.pos.y - sideLength});
hero.command(illumina, "move", {x: celadia.pos.x + sideLength, y: celadia.pos.y - sideLength});
```

#33. Bits And Trits

Level Overview and Solutions

Intro

Brawlers incoming!

Command the robot by converting base-10 numbers to base-2 and base-3!

Default Code

```
// Incoming Ogre Brawlers!
// Make use of a Robot Walker to dispatch these enemies.
// The Robot Walker requires commands as strings.
// The first part will the enemy's health in ternary.
// The second part will be the enemy's type as binary.

function toTernary(number) {
    // Start with an empty string.
    var string = "";
    // Then, while the number isn't zero:
    while(number !== 0) {
        // We grab the remainder of our number.
        var remainder = number % 3;
        // This is our iterator method. 'number' decrements here.
        number = (number - remainder) / 3;
        // Append the string to the remainder.
        string = remainder + string;
    }
    // Finally we want to return our constructed string.
    return string;
}

function toBinary(number) {
    var string = "";
    // Go through the steps again:

        // Get remainder, decrement, and append string.

    // Remember that binary is another way of saying '2'!

    return string;
}

while(true) {
    var enemies = hero.findEnemies();
    var dangerous = findMostDangerous(enemies);
    if(dangerous) {
        // The way the robot takes commands is in the form of:
        // ternary(enemyHealth) + " " + binary(enemyType)
        hero.say(toTernary(dangerous.health) + " " + toBinary(dangerous.type));
    }
}

// In this level the Ogre Brawlers are more powerful if they have more health.
function findMostDangerous(enemies) {
    var mostDangerous = null;
    var mostHealth = -Infinity;
    for(var i = 0; i < enemies.length; i++) {
        var enemy = enemies[i];
        if(enemy.health > mostHealth) {
            mostDangerous = enemy;
            mostHealth = enemy.health;
        }
    }
    return mostDangerous;
}
```

Overview

You will need to convert our normal decimal number system into something this robot can understand!

We use the decimal system, where *dec* means 10, like a decade (10 years). There are other number systems out there such as the binary (bicentennial, or 200 years) and trinary system: *bi* means 2, *tri* means 3.

We call each number system as a *base*, so *base 10* means decimal, while *base 2* means binary.

Observe the various ways to write the number 9 in various bases:

```
Base 10: 9
Base  9: 10
Base  8: 11
Base  7: 12
Base  6: 13
Base  5: 14
Base  4: 21
Base  3: 100
Base  2: 1001
```

The method we use to convert a number to another base is to divide by the desired base and observe the remainder:

```
Convert 9 base 10 to base 3
  9 / 3 = 3, with 0 remainder
 3 / 3 = 1, with 0 remainder
1 / 3 = 0, with 1 remainder
Notice that the remainders equal to 100!

Convert 9 base 10 to base 2
   9 / 2 = 4, with 1 remainder
  4 / 2 = 2, with 0 remainder
 2 / 2 = 1, with 0 remainder
1 / 2 = 0, with 1 remainder
Again, 1001 is the same as the provided answer above!
```

But how can you trust the numbers provided above are the actual binary and trinary representations of 9?

How to count in a base number system: In the binary system there are only 2 digits: (0, 1) In the trinary system there are only 3 digits: (0, 1, 2) In the decimal system there are 10 digits: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) In the hexadecimal system there are 16 digitS: (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f) Yes! Letters, too!

Now to count in binary, remeber you cannot ever reach the digit 2!

```
Base 10     Base 2
      0          0
      1          1
      2         10
      3         11
      4        100
      5        101
      6        110
      7        111
      8       1000
      9       1001
     10       1010
     11       1011
     12       1100
```

See? It matches! Now command the robot with this knowledge.

Bits And Trits Solution

```
// Incoming Ogre Brawlers!
// Make use of a Robot Walker to dispatch these enemies.
// The Robot Walker requires commands as strings.
// The first part will the enemy's health in ternary.
// The second part will be the enemy's type as binary.

function toTernary(number) {
    // Start with an empty string.
    var string = "";
    // Then, while the number isn't zero:
    while(number !== 0) {
        // We grab the remainder of our number.
        var remainder = number % 3;
        // This is our iterator method. 'number' decrements here.
        number = (number - remainder) / 3;
        // Append the string to the remainder.
        string = remainder + string;
    }
    // Finally we want to return our constructed string.
    return string;
}

function toBinary(number) {
    var string = "";
    // Go through the steps again:
        // Get remainder, decrement, and append string.
    // Remember that binary is another way of saying '2'!
    while(number !== 0) {
        var remainder = number % 2;
        number = (number - remainder) / 2;
        string = remainder + string;
    }
    return string;
}

while(true) {
    var enemies = hero.findEnemies();
    var dangerous = findMostDangerous(enemies);
    if(dangerous) {
        // The way the robot takes commands is in the form of:
        // ternary(enemyHealth) + " " + binary(enemyType)
        hero.say(toTernary(dangerous.health) + " " + toBinary(dangerous.type));
    }
}

// In this level the Ogre Brawlers are more powerful if they have more health.
function findMostDangerous(enemies) {
    var mostDangerous = null;
    var mostHealth = -Infinity;
    for(var i = 0; i < enemies.length; i++) {
        var enemy = enemies[i];
        if(enemy.health > mostHealth) {
            mostDangerous = enemy;
            mostHealth = enemy.health;
        }
    }
    return mostDangerous;
}
```