Computer Science 2

JavaScript

Introduces arguments, variables, if statements, and arithmetic.

#1. Defense of Plainswood

Level Overview and Solutions

Intro

Move your mouse cursor over a spot to find the `x` and `y` numbers of the location.

Remember how to build:

```
hero.buildXY("fence", 40, 52)
```

Default Code

```
// Build two fences on the marks to keep the villagers safe!
// Hover your mouse over the world to get X,Y coordinates.
```

Overview

Use your `buildXY` skills to build `"fence"` s and block out the ogres!

Remember to hover over the level map to find `x` and `y` coordinates for your `buildXY` method. In this case, you want to build on the X marks at `40, 52` and `40, 20`.

It is much, much easier to do this level by building type `"fence"` than by building type `"fire-trap"`.

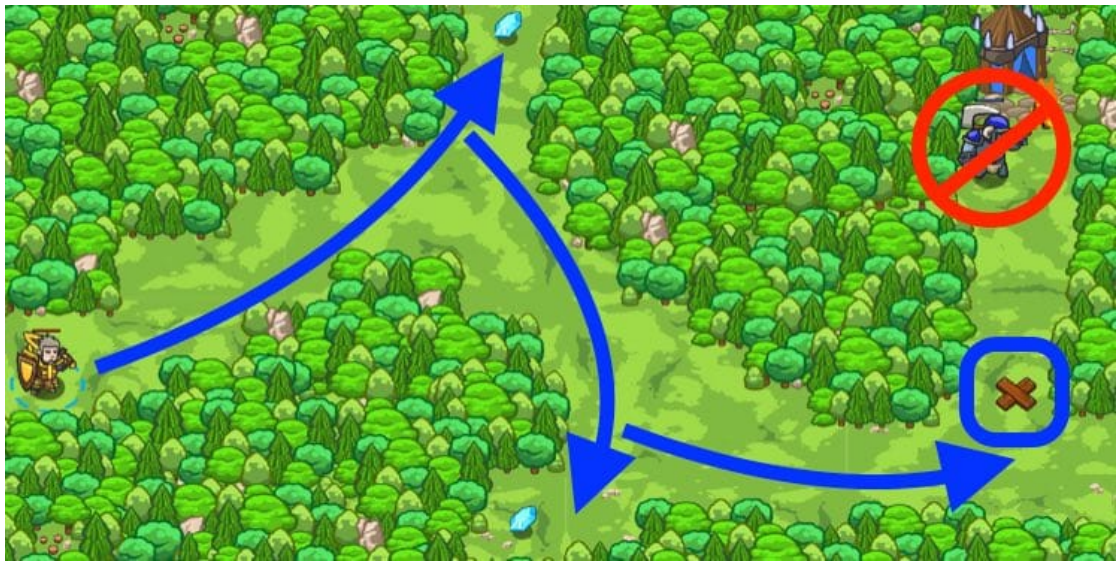
Defense of Plainswood Solution

```
// Build two fences on the marks to keep the villagers safe!
// Hover your mouse over the world to get X,Y coordinates.
hero.buildXY("fence", 40, 53);
hero.buildXY("fence", 40, 21);
```

#2. Course: Winding Trail

Level Overview and Solutions

Intro



Just like building, you can now move to any position on the map using `moveXY(x, y)`. Mouse over where you want to go to get the `x` and `y` coordinates, then use:

```
hero.moveXY(30, 30);
```

to move there.

Default Code

```
// Go to the end of the path and build a fence there.
// Use your moveXY(x, y) function.

hero.moveXY(34, 45);
```

Overview

Forget those old simple `moveRight` boots!

Your new digs let you `moveXY` for continuous movement, wherever you want to go. They even have pathfinding built in. Sweet, huh?

Just like with `buildXY`, you can hover over the level to find `x` and `y` coordinates for you to move to.

Move to each gem in turn, then stop the ogre from getting you by building a fence on the X marker!

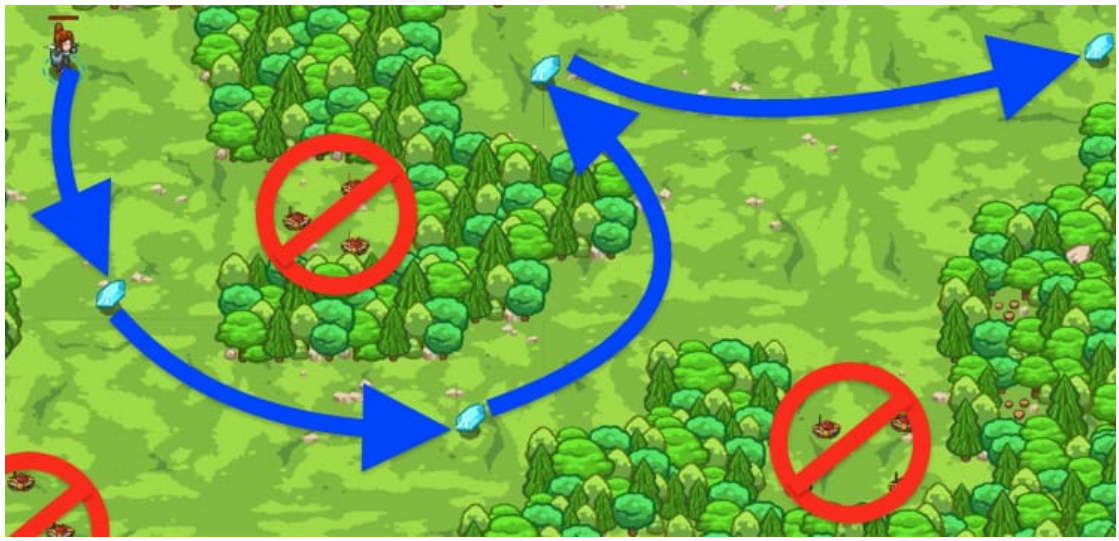
Course: Winding Trail Solution

```
// Go to the end of the path and build a fence there.
// Use your moveXY(x, y) function.

hero.moveXY(36, 59);
hero.moveXY(37, 12);
hero.moveXY(66, 17);
hero.buildXY("fence", 71, 24);
```

#3. One Wrong Step

Level Overview and Solutions

Intro



Code can be changed! One's destiny is not defined merely by the sample code provided.

```
hero.moveXY(26, 43); // Danger ahead! Erase the coordinates and move towards the gems instead.
```

Default Code

```
// The hero is all confused!
// Correct their path so they don't walk on the mines.

hero.moveXY(26, 43);
hero.moveXY(6, 15);
hero.moveXY(61, 22);
hero.moveXY(57, 66);
```

Overview

Navigate through the forest and gather gems on your way!

Watch out! The default code was written by an ogre that wants you to run into mines!

Edit the coordinates so the hero moves to each gem.

You must collect each gem to win!

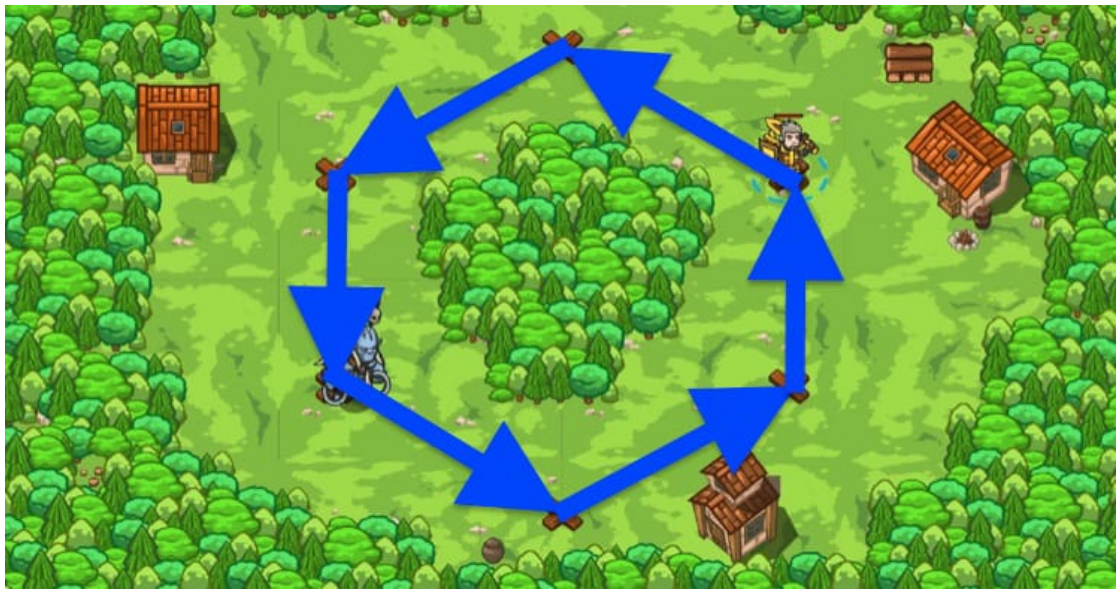
One Wrong Step Solution

```
// The hero is all confused!
// Correct their path so they don't walk on the mines.

hero.moveXY(11, 35);
hero.moveXY(35, 24);
hero.moveXY(40, 55);
hero.moveXY(77, 57);
```

#4. Forest Evasion

Level Overview and Solutions

Intro



Use a `while-true` loop and `moveXY` to stay out of the eyesight of the Headhunter.

Default Code

```
// There is a headhunter in the area!
// Move around the forest to avoid his line-of-sight.

while(true) {
    hero.moveXY(56,44);
    hero.moveXY(40,56);
    // Use moveXY to keep moving around the forest to survive.

}
```

Overview

The deadly headhunter is coming for your hero's head! Use `moveXY` to move around the group of trees and stay out of sight.

The headhunter can't attack you if he doesn't see you!

Forest Evasion Solution

```
// There is a headhunter in the area!
// Move around the forest to avoid his line-of-sight.

while(true) {
    hero.moveXY(56,44);
    hero.moveXY(40,56);
    // Use moveXY to keep moving around the forest to survive.
    hero.moveXY(24,44);
    hero.moveXY(24,24);
    hero.moveXY(40,12);
    hero.moveXY(56,24);
}
```

#5. Woodland Cubbies

Level Overview and Solutions

Intro

Check the wooded cubbies for enemies, but beware! There may not always be an enemy to attack.

Use an `if-statement` to check the **existence** of an enemy.

```
var enemy = hero.findNearestEnemy();
if(enemy) {
    // Attack!
}
```

Default Code

```
// Navigate through the woods, but be on the lookout!
// These forest cubbies may contain ogres!

hero.moveXY(19, 33);
var enemy = hero.findNearestEnemy();
// The if-statement will check if a variable has an ogre.
if(enemy) {
    hero.attack(enemy);
    hero.attack(enemy);
}

hero.moveXY(49, 51);
var enemy = hero.findNearestEnemy();
if(enemy) {
    // Attack the enemy here:

}

hero.moveXY(58, 14);
var enemy = hero.findNearestEnemy();
// Use an if-statement to check if enemy exists:

    // If enemy exists, attack it:
```

Overview

if-statements

`if-statements` are used to control the **flow** of a program. They can be used to check if a certain **condition** is `true`.

`if-statements` are like `while`, but instead of `true`, a `conditional` should be checked against.

Commonly, `if` can be used to check if a unit **exists** by adding it after the `if`.

Examples:

```
var enemy = hero.findNearestEnemy();
if(enemy) {
    // The enemy exists.
    // Probably should attack here!
}
// This always happens whether or not there is an enemy!
```

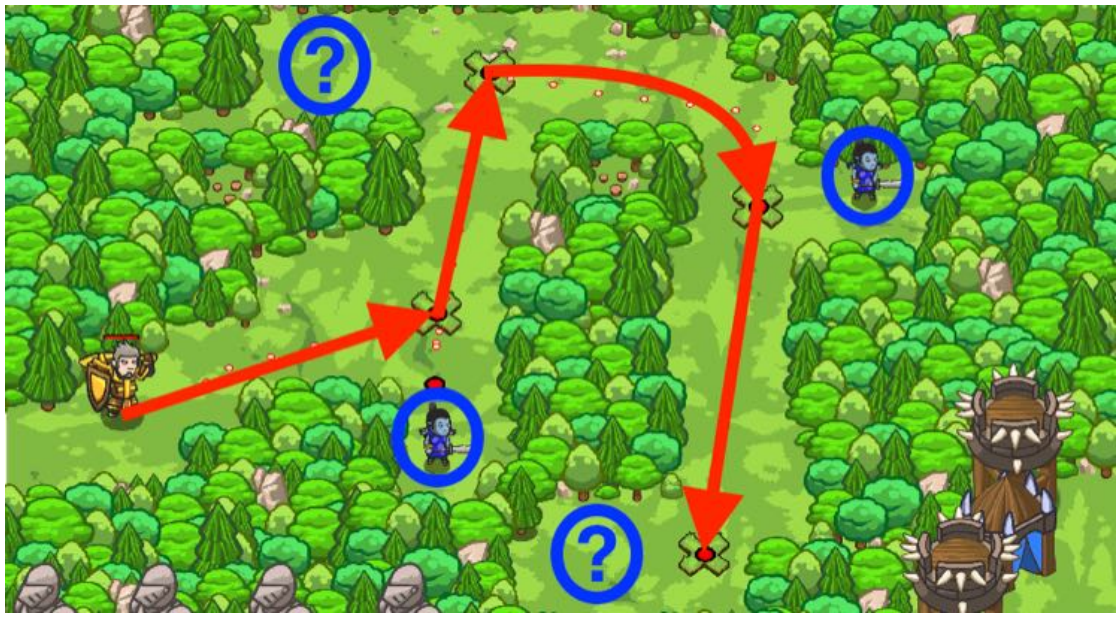
Woodland Cubbies Solution

```
// Navigate through the woods, but be on the lookout!
// These forest cubbies may contain ogres!

hero.moveXY(19, 33);
var enemy = hero.findNearestEnemy();
// The if-statement will check if a variable has an ogre.
if(enemy) {
    hero.attack(enemy);
    hero.attack(enemy);
}

hero.moveXY(49, 51);
var enemy = hero.findNearestEnemy();
if(enemy) {
    // Attack the enemy here:
    hero.attack(enemy);
    hero.attack(enemy);
}

hero.moveXY(58, 14);
var enemy = hero.findNearestEnemy();
// Use an if-statement to check if enemy exists:
if(enemy) {
    // If enemy exists, attack it:
    hero.attack(enemy);
    hero.attack(enemy);
}
```

#5a. Backwoods Ambush

Level Overview and Solutions

Intro



Use `moveXY` to patrol the forest.

You now have access to the powerful `if-statements`. Check your toolbar in the `lower right` for the `Programmaticon II` for extra information.

`if-statements` are a fundamental tool for programmers. Create them by typing:

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

Default Code

```
hero.moveXY(24, 42);
var enemy = hero.findNearestEnemy();
if(enemy) {
    hero.attack(enemy);
    hero.attack(enemy);
}

hero.moveXY(27, 60);
enemy = hero.findNearestEnemy();
if(enemy) {
    // Attack the enemy if it exists!

}

hero.moveXY(42, 50);
enemy = hero.findNearestEnemy();
// Use an if-statement to check if an enemy exists.

    // Attack the enemy if it exists!


hero.moveXY(39, 24);
// Find the nearest enemy:

// Check if the enemy exists:

    // Attack the enemy if it exists!
```

Overview

Your new Programmaticon II grants you the ability to use **if-statements**.

An **if-statement** says, **if** some condition is true, then run some code (if it's not true then don't run the code!)

To complete this level, you should move to each of the X marks with `moveXY`.

At each X spot, there may or may not be an ogre (the ogres are spawned randomly each time you press the Submit button!).

So use `findNearestEnemy` and `if` statements to determine if an ogre is at each spot, like this:

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

When you use an `if` statement this way, you won't get an error by trying to attack an enemy when there is no one there!

Backwoods Ambush Solution

```
hero.moveXY(24, 42);
var enemy = hero.findNearestEnemy();
if(enemy) {
    hero.attack(enemy);
    hero.attack(enemy);
}

hero.moveXY(27, 60);
enemy = hero.findNearestEnemy();
if(enemy) {
    // Attack the enemy if it exists!
    hero.attack(enemy);
    hero.attack(enemy);
}

hero.moveXY(42, 50);
enemy = hero.findNearestEnemy();
// Use an if-statement to check if an enemy exists.
if(enemy) {
    // Attack the enemy if it exists!
    hero.attack(enemy);
    hero.attack(enemy);
}
hero.moveXY(39, 24);
// Find the nearest enemy:
enemy = hero.findNearestEnemy();
// Check if the enemy exists:
if(enemy) {
    // Attack the enemy if it exists!
    hero.attack(enemy);
    hero.attack(enemy);
}
```

#6. Patrol Buster

Level Overview and Solutions

Intro

You now have access to the powerful `if-statements`. Check your toolbar in the `lower right` for the `Programmaticon II` for extra information.

`if-statements` are a fundamental tool for programmers. Create them by typing:

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

Default Code

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        // If there is an enemy, attack it!

    }
}
```

Overview

Your new Programmaticon II grants you the ability to use `if-statement`s. They let you run code only if a certain condition is true.

In this level, you want to attack the nearest enemy, but only if there is an enemy. Use an **if-statement** with `enemy` as the condition to do that.

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

Remember to hover over the `if/else` and read the example code in the **lower right** to see what the syntax should be.

Patrol Buster Solution

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        // If there is an enemy, attack it!
        hero.attack(enemy);
    }
}
```

#6a. Patrol Buster A

Level Overview and Solutions

Intro

You now have access to the powerful `if-statements`. Check your toolbar in the `lower right` for the `Programmaticon II` for extra information.

`if-statements` are a fundamental tool for programmers. Create them by typing:

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

Default Code

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        // If there is an enemy, attack it!

    }
}
```

Overview

Your new Programmaticon II grants you the ability to use `if-statement`s. They let you run code only if a certain condition is true.

In this level, you want to attack the nearest enemy, but only if there is an enemy. Use an **if-statement** with `enemy` as the condition to do that.

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

Remember to hover over the `if/else` and read the example code in the **lower right** to see what the syntax should be.

Patrol Buster A Solution

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        // If there is an enemy, attack it!
        hero.attack(enemy);
    }
}
```

#7. Picnic Buster

Level Overview and Solutions

Intro

Protect the picnickers by using `if` statements to figure out when there are ogres to attack.

`if` statements are a fundamental tool for programmers. They look like this:

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

Default Code

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    // If there is an enemy, attack it!

}
```

Overview

Just as in "Patrol Buster," you want to attack the nearest enemy, but only if there is one. Use an `if` statement with `enemy` as the condition to do that.

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

Remember to hover over the `if/else` and read the example code in the **lower right** to see what the syntax should be.

Picnic Buster Solution

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    // If there is an enemy, attack it!
    if (enemy) {
        hero.attack(enemy);
    }
}
```

#7a. Eagle Eye

Level Overview and Solutions

Intro

A Griffin pal has arrived to help! It will call out the ogres as they arrive.

There won't always be an ogre, so you should use an `if-statement` to check if an enemy exists before trying to attack it.

```
var enemy = hero.findNearestEnemy();
if(enemy) {
    // There IS an enemy nearby!
}
```

Default Code

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    // If there is an enemy, attack it!

}
```

Overview

You must use an `if-statement` to check if an enemy exists, so the hero doesn't try to attack someone that isn't there!

For example, a fisherman would check if a fish is caught every time they cast their line:

```
var fish = fisherman.castLine();
if(fish) {
    fisherman.store(fish);
}
```

Imagine trying to store nothing in a pail! It's confusing to think about.

Eagle Eye Solution

```
// Remember that enemies may not yet exist.
while (true) {
    var enemy = hero.findNearestEnemy();
    // If there is an enemy, attack it!
    if (enemy) {
        hero.attack(enemy);
    }
}
```

#8. If-stravaganza

Level Overview and Solutions

Intro

Defend against ogres using an `if-statement` to check if here is an ogre nearby!

Do **not** attack if there is no munchkin around! You'll confuse the hero.

Default Code

```
// Defeat the ogres from within their own camp!

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement to check if an enemy exists:

        // Attack the enemy if it exists:

}
```

Overview

If-statements

`if-statements` are used to only perform actions if something (the `conditional` of the `if-statement`) is `true`.

```
if(conditional) {
    // Code inside here executes if "condition" is true.
}
```

The `conditional` can be used to check existence, whether something is close or far, or the type of a unit.

In this level, you need to check whether or not a munchkin exists when searching, and to attack it if it does exist.

```
var tree = hero.findNearestTree();
var fruit = hero.pluckFruit(tree); // A tree doesn't always have fruit.
// The fruit variable potentially holds information about a tree's fruit.
if(fruit) {
    // If the variable fruit holds information, then code inside the if-statement occurs.
    hero.eat(fruit);
}
```

If-stravaganza Solution

```
// Defeat the ogres from within their own camp!

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement to check if an enemy exists:
    if(enemy) {
        // Attack the enemy if it exists:
        hero.attack(enemy);
    }
}
```

#9. Village Guard

Level Overview and Solutions

Intro

Patrol the village's entry points and use an `if-statement` to check for enemies.

Default Code

```
// Patrol the village entrances.
// If you find an enemy, attack it.
while(true) {
    hero.moveXY(35, 34);
    var left = hero.findNearestEnemy();
    if (left) {
        hero.attack(left);
        hero.attack(left);
    }
    // Now move to the right entrance.

    // Find the right enemy.
    // Use "if" to attack if there is a right enemy.

}
```

Overview

You can do this level with two **if-statements**.

The first one, with `left`, is in the default code as an example, so reload the sample code if you get off track.

Move to the X on the right, then define a `right` variable with your `findNearestEnemy` method. Then, write an **if-statement** to check if `right` exists. If there is an enemy, attack it!

Make sure that you define the `right` variable when you would be able to see an enemy coming on the right.

Village Guard Solution

```
// Patrol the village entrances.
// If you find an enemy, attack it.
while(true) {
    hero.moveXY(35, 34);
    var left = hero.findNearestEnemy();
    if(left) {
        hero.attack(left);
        hero.attack(left);
    }
    // Now move to the right entrance.
    hero.moveXY(60, 31);
    // Find the right enemy.
    // Use "if" to attack if there is a right enemy.
    var right = hero.findNearestEnemy();
    if(right) {
        hero.attack(right);
        hero.attack(right);
    }
}
```

#10. Thornbush Farm

Level Overview and Solutions

Intro

Patrol around the three entrances, and build a `"fire-trap"` at each X if you see an enemy.

Build a `"fire-trap"` the same way you build a `"fence"`, just with a different string:

```
hero.buildXY("fire-trap", 20, 20);
```

Default Code

```
// Patrol the village entrances.
// Build a "fire-trap" when you see an ogre.
// Don't blow up any peasants!

while(true) {
    hero.moveXY(43, 50);
    var top = hero.findNearestEnemy();
    if (top) {
        hero.buildXY("fire-trap", 43, 50);
    }

    hero.moveXY(25, 34);
    var left = hero.findNearestEnemy();
    // Check if left exists.

        // Build a trap at 25, 34 if the enemy exists.


    hero.moveXY(43, 20);
    // Set a variable for the bottom enemy.

    // Check if the bottom enemy exists.

        // Build a trap if an enemy exists.

}
```

Overview

Ogres are coming from the top, left, and bottom, so you need three sets of commands in your **loop**: one for `top`, one for `left`, and one for `bottom`.

Write the `left` and `bottom` code based on the `top` sample code.

Make sure that in each set of commands, you:

1. First, `moveXY` to the X marker
2. Define a new enemy variable with `findNearestEnemy` **after** you get to the marker
3. Write an if statement: *if* there is an enemy, *then* build a `"fire-trap"` on the X marker

After that, your loop will repeat to patrol all three entrances several times.

You only want to build a fire trap if you see an ogre coming, because otherwise a peasant will try to get into the village only to be blown to smithereens by your fire trap!

If you are getting stuck, look very closely at the `top` part to make sure your code is formatted the same way for `left` and `bottom`.

Thornbush Farm Solution

```
// Patrol the village entrances.
// Build a "fire-trap" when you see an ogre.
// Don't blow up any peasants!

while(true) {
    hero.moveXY(43, 50);
    var top = hero.findNearestEnemy();
    if(top) {
        hero.buildXY("fire-trap", 43, 50);
    }

    hero.moveXY(25, 34);
    var left = hero.findNearestEnemy();
    if(left) {
        hero.buildXY("fire-trap", 25, 34);
    }

    hero.moveXY(43, 20);
    var bottom = hero.findNearestEnemy();
    if(bottom) {
        hero.buildXY("fire-trap", 43, 20);
    }
}
```

#11. Back to Back

Level Overview and Solutions

Intro

Expand your code's possibilities by using `else`!

`else` is just like `if`, except it contains the code that should run when the `if` condition is `false`:

```
if(enemy) {
    // This happens when there is an enemy.
    hero.attack(enemy);
} else {
    // This happens when there is no enemy.
    hero.say("I don't see an enemy!");
}
```

Default Code

```
// Stay in the middle and defend!

while(true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        // Either attack the enemy...

    }
    else {
        // ... or move back to your defensive position.

    }
}
```

Overview

This level introduces the `else` part of `if/else`.

When you add an `else` clause, you choose what to do both when the condition is true and when it is not true.

So you can say, *if* there is an enemy, *then* attack it, *else* move to the X.

To show you how it works, the `if` and the `else` are set up for you, and you need to put in the `attack` and `moveXY` methods so that your hero attacks enemies on sight, but when there are no enemies, moves back to the X marker to defend the peasants.

Make sure you get the coordinates for the X marker correct, or you might not be able to defend both your peasants in time.

Back to Back Solution

```
// Stay in the middle and defend!

while(true) {
    var enemy = hero.findNearestEnemy();
    // Either attack the enemy...
    if(enemy) {
        hero.attack(enemy);
        hero.attack(enemy);
    }
    // ... or move back to your defensive position.
    else {
        hero.moveXY(40, 34);
    }
}
```

#12. Ogre Encampment

Level Overview and Solutions

Intro

If there is an enemy `attack` it, otherwise `attack` the `"Chest"`.

Default Code

```
// If there is an enemy, attack it.
// Otherwise, attack the chest!

while(true) {
    // Use if/else.

    hero.attack("Chest");

}
```

Overview

For this level, you'll need to use both `if` and `else`. Remember that the `else` block executes when the `if` condition is not true.

When the ogres attack you, you want to fight back, but when there are no ogres, you can keep attacking the `"Chest"` to open it. So in your `if` condition, check whether there is an enemy. If there is, attack it. Else, attack the `"Chest"`.

To remember the if/else syntax, either hover over the `if/else` examples in the lower right, from your Programmaticon II.

Ogre Encampment Solution

```
// If there is an enemy, attack it.
// Otherwise, attack the chest!

while(true) {
    // Use if/else.
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        hero.attack(enemy);
    } else {
        hero.attack("Chest");
    }
}
```

#13. Woodland Cleaver

Level Overview and Solutions

Intro

You have a new sword with the method `cleave()`, and a new watch with the method `isReady()`.

`cleave()` is a special attack that hits all nearby enemies, but it can only be used every so often.

Before you `cleave()`, check if it's ready:

```
if(hero.isReady("cleave")) {
    hero.cleave(enemy);
} else {
    hero.attack(enemy);
}
```

Default Code

```
// Use your new "cleave" skill as often as you can.

hero.moveXY(23, 23);
while(true) {
    var enemy = hero.findNearestEnemy();
    if (hero.isReady("cleave")) {
        // Cleave the enemy!

    }
    else {
        // Else (if cleave isn't ready), do your normal attack.

    }
}
```

Overview

The woods are swarming with ogre munchkins, but you have a new Long Sword, and its `cleave` ability will make short work of them! `cleave` hits every enemy within ten meters of your hero.

Special abilities like `cleave` have cooldown periods, which means you can't use them all the time. (You can only cleave every ten seconds.) You need to check if they are ready to use first. Fortunately, your Sundial Wristwatch gives you the `isReady` method. It tells you whether special abilities are ready to be used yet.

Putting everything together, your code should go like this:

- loop

- find an enemy
- *if* `"cleave"` is ready, *then*
  - cleave the enemy
- *else*
  - attack the enemy

Hover over the `isReady` and `cleave` documentation in the lower right to see the syntax for how to use them.

Woodland Cleaver Solution

```
// Use your new "cleave" skill as often as you can.

hero.moveXY(23, 23);
while(true) {
    var enemy = hero.findNearestEnemy();
    if(hero.isReady("cleave")) {
        // Cleave the enemy!
        hero.cleave(enemy);
    } else {
        // Else (if cleave isn't ready), do your normal attack.
        hero.attack(enemy);
    }
}
```

#14. Elseweyr

Level Overview and Solutions

Intro

Wield your new long sword like a pro! Use `isReady()` to check if you can `"cleave"` and then `cleave()`!

Use an `else` statement to ensure the hero defends themself while the munchkins rampage around.

Default Code

```
// Your cleave is on a 10 second cooldown.
// Use an else-statement to defend yourself while recharging.

while(true) {
    var enemy = hero.findNearestEnemy();
    if(hero.isReady("cleave")) {
        hero.cleave();
    }
    // Write else: to do something when cleave isn't ready:

        // Be sure to attack the enemy:

}
```

Overview

`else` can be used to perform actions when an `if-statement` isn't `true`. `else`s can *ONLY* be used after an `if-statement` within the same **scope**. In otherwords, they must be on the same line as the previous `if-statement`.

When reading an `else`, consider it to be all other options other than what the connected `if-statement` was.

```
var enemy = hero.findNearestEnemy();
// Check if an enemy exists
if(enemy) {
    // There is an enemy, so attack it.
    hero.attack(enemy);
}
// Otherwise (or else), there isn't an enemy.
else {
    // There is NOT an enemy, so relax.
    hero.say("I'm safe.");
}
```

Elseweyr Solution

```
// Your cleave is on a 10 second cooldown.
// Use an else-statement to defend yourself while recharging.

while(true) {
    var enemy = hero.findNearestEnemy();
    if(hero.isReady("cleave")) {
        hero.cleave();
    }
    // Write else: to do something when cleave isn't ready:
    else {
        // Be sure to attack the enemy:
        hero.attack(enemy);
    }
}
```

#15. Backwoods Standoff

Level Overview and Solutions

Intro

`if cleave isReady`, use it!

`else attack` the nearest enemy!

Default Code

```
// Munchkins are attacking!
// The swarms will come at regular intervals.
// Whenever you can, cleave to clear the mass of enemies.

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement with isReady to check "cleave":

        // Cleave!

    // Else, if cleave is not ready:

        // Attack the nearest ogre!

}
```

Overview

The munchkins are attacking! Use an if/else statement to perform an alternate action while `cleave` is on cooldown.

`if "cleave" isReady`, then `cleave`!

`else attack` the nearest `enemy`.

The using of `else` helps you to avoid a problem when your hero tries to attack the enemy which was a target for `cleave`.

Backwoods Standoff Solution

```
// Munchkins are attacking!
// The swarms will come at regular intervals.
// Whenever you can, cleave to clear the mass of enemies.

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement with isReady to check "cleave":
    if(hero.isReady("cleave")) {
        // Cleave!
        hero.cleave();
    }
    // Else, if cleave is not ready:
    else {
        // Attack the nearest ogre!
        hero.attack(enemy);
    }
}
```

#15a. Backwoods Standoff A

Level Overview and Solutions

Intro

`if cleave isReady`, use it!

`else attack` the nearest enemy!

Default Code

```
// Munchkins are attacking!
// The swarms will come at regular intervals.
// Whenever you can, cleave to clear the mass of enemies.

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement with isReady to check "cleave":

        // Cleave!

    // Else, if cleave is not ready:

        // Attack the nearest ogre.

}
```

Overview

The munchkins are attacking! Use an if/else statement to perform an alternate action while `cleave` is on cooldown.

`if "cleave" isReady`, then `cleave`!

`else attack` the nearest `enemy`.

Backwoods Standoff A Solution

```
// Munchkins are attacking!
// The swarms will come at regular intervals.
// Whenever you can, cleave to clear the mass of enemies.

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement with isReady to check "cleave":
    if(hero.isReady("cleave")) {
        // Cleave!
        hero.cleave();
    }
    // Else, if cleave is not ready:
    else {
        // Attack the nearest ogre.
        hero.attack(enemy);
    }
}
```

#15b. Backwoods Standoff B

Level Overview and Solutions

Intro

`if cleave isReady`, use it!

`else attack` the nearest enemy!

Default Code

```
// Munchkins are attacking!
// The swarms will come at regular intervals.
// Whenever you can, cleave to clear the mass of enemies.

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement with isReady to check "cleave":

        // Cleave!

    // Else, if cleave is not ready:

        // Attack the nearest ogre!

}
```

Overview

The munchkins are attacking! Use an if/else statement to perform an alternate action while `cleave` is on cooldown.

`if "cleave" isReady`, then `cleave`!

`else attack` the nearest `enemy`.

Backwoods Standoff B Solution

```
// Munchkins are attacking!
// The swarms will come at regular intervals.
// Whenever you can, cleave to clear the mass of enemies.

while(true) {
    var enemy = hero.findNearestEnemy();
    // Use an if-statement with isReady to check "cleave":
    if(hero.isReady("cleave")) {
        // Cleave!
        hero.cleave();
    }
    // Else, if cleave is not ready:
    else {
        // Attack the nearest ogre!
        hero.attack(enemy);
    }
}
```

#16. Range Finder

Level Overview and Solutions

Intro

Use `distanceTo()` to find the range to each enemy, and use `say()` to call the range for your artillery.

Then sit back and watch the fireworks.

Default Code

```
// Ogres are scouting the forest!
// Use the distanceTo method to find where the enemies are.
// Say the distance for each enemy to tell the artillery where to fire!

var enemy1 = "Gort";
var distance1 = hero.distanceTo(enemy1);
hero.say(distance1);

var enemy2 = "Smasher";
var distance2 = hero.distanceTo(enemy2);
// Say the distance2 variable!

// Find and say the distance to the rest of the enemies:
// Don't shoot at your friends!
var friend3 = "Charles";

var enemy4 = "Gorgnub";
```

Overview

You've been asked to test special glasses that can see through trees! This time, you don't need to go out and deal with the ogres personally.

Your artillery can't sight through the trees, so use `distanceTo()` and `say()` to call out the range to each target.

Be careful, though! There are peaceful woodsmen living in these woods.

Range Finder Solution

```
// Ogres are scouting the forest!
// Use the distanceTo method to find where the enemies are.
// Say the distance for each enemy to tell the artillery where to fire!

var enemy1 = "Gort";
var distance1 = hero.distanceTo(enemy1);
hero.say(distance1);

var enemy2 = "Smasher";
var distance2 = hero.distanceTo(enemy2);
// Say the distance2 variable!
hero.say(distance2);
// Find and say the distance to the rest of the enemies:
// Don't shoot at your friends!
var friend3 = "Charles";
var distance3 = hero.distanceTo(friend3);
//hero.say(distance3);
var enemy4 = "Gorgnub";
var distance4 = hero.distanceTo(enemy4);
hero.say(distance4);
```

#17. Peasant Protection

Level Overview and Solutions

Intro

If an enemy gets too close to you, `attack` it! Otherwise, `moveXY` to the X to stay close to the peasant.

Remember, you can find the distance using:

```
var distance = hero.distanceTo(enemy);
```

Default Code

```
while(true) {
    var enemy = hero.findNearestEnemy();
    var distance = hero.distanceTo(enemy);
    if (distance < 10) {
        // Attack if they get too close to the peasant.

    }
    // Else, stay close to the peasant! Use else.


}
```

Overview

You can use `distanceTo` to measure the distance, in meters, to a target. In this level, you'll use that to make sure you stay close to vulnerable peasant Victor.

You can see a new piece of syntax here, the **less-than** operator: `<`

You can read it like this: *if* the distance is *less than* 10 meters, *then* attack the enemy, *else* move back to the X marker.

Fill in the `else` to move back to the X so that no ogres can get to Victor while you're far away.

Tip: make sure that you are moving to the correct defensive position–the X is at `{x: 40, y: 37}`.

Peasant Protection Solution

```
while(true) {
    var enemy = hero.findNearestEnemy();
    var distance = hero.distanceTo(enemy);
    if(distance < 10) {
        // Attack if they get too close to the peasant.
        hero.attack(enemy);
    }
    // Else, stay close to the peasant! Use else.
    else {
        hero.moveXY(40, 37);
    }
}
```

#18. Munchkin Swarm

Level Overview and Solutions

Intro

The ingredients to beat this level are: `if/else`, `distanceTo()`, `<`, `cleave()`, `while-true loop`, and `attack("Chest")`. Put them all together to break the chest and survive the Munchkin onslaught.

Default Code

```
while(true) {
    // Check the distance to the nearest enemy.

    // If it comes closer than 10 meters, cleave it!

    // Else, attack the "Chest" by name.

}
```

Overview

In this level, you put together everything you've learned over the past few levels in order to use `if/else`, `distanceTo`, `<`, and `cleave` to defeat vast numbers of ogre munchkins while looting a giant treasure chest.

These munchkins have become suitably terrified of you and your mighty Long Sword, so they will only approach when there are a lot of them together in a pack. Check the distance to the nearest munchkin and only `cleave` if the munchkin is closer than ten meters. Use an `else` clause to attack the `"Chest"` otherwise.

Tip: make sure to use a **while-true loop**.

Tip: you'll know that your distance check is working when your hero never chases any munchkins away from the chest.

Munchkin Swarm Solution

```
while(true) {
    // Check the distance to the nearest enemy.
    var enemy = hero.findNearestEnemy();
    var distance = hero.distanceTo(enemy);
    // If it comes closer than 10 meters, cleave it!
    if(distance < 10) {
        hero.cleave(enemy);
    } else {
        // Else, attack the "Chest" by name.
        hero.attack("Chest");
    }
}
```

#19. Maniac Munchkins

Level Overview and Solutions

Intro

Break open a chest while being attacked by groups of munchkins and certain, particularly angry, munchkins.

```
if (condition1) {
    // This only happens if condition1 is true.
} else if (condition2) {
    // This only happens if condition1 is false and condition2 is true.
} else {
    // This only happens if both condition1 and condition2 are false.
}
```

Default Code

```
// Another chest in the field for the hero to break open!
// Attack the chest to break it open.
// Some munchkins won't stand idly by while you attack it!
// Defend yourself when a munchkin gets too close.

while(true) {
    var enemy = hero.findNearestEnemy();
    var distance = hero.distanceTo(enemy);
    if(hero.isReady("cleave")) {
        // First priority is to cleave if it's ready:

    } else if(distance < 5) {
        // Attack the nearest munchkin that gets too close:

    } else {
        // Otherwise, try to break open the chest:

    }
}
```

Overview

In this level, the munchkins will periodically attack without the support of their comrades!

Use `cleave` to defeat any groups of munchkins that get close, but only use it when it's off cooldown! `isReady` will help with that.

```
if(hero.isReady("cleave")) {
    // This will only happen when the hero's "cleave" is ready to be used.
}
```

Check if munchkins get to close using the `distanceTo` method. Remember that `distanceTo` finds a `number` between the hero and the argument.

Note that the `less-than` sign, or `<` is only useful at comparing two `numbers`! `hero.findNearestEnemy()` returns an enemy, not a number! `hero.isReady()` returns a `true` or `false` value, not a number! Be sure to only use `<` when comparing 2 numbers, like `5`, or `hero.distanceTo(enemy)`.

```
var enemy = hero.findNearestEnemy()
var distance = hero.distanceTo(enemy)
if(hero.isReady("cleave") {
// else if is a special term! It tells the hero to not do the next part if the first part was true, or
    tells them to do the second part if the first part wasn't true.
} else if(distance < 5) {
    // This will only happen when the enemy is closer than 5 meters and cleave isn't ready.
}
```

Finally, if cleave isn't ready and the nearest munchkin is more than 5 meters away, you're free to attack the chest!

```
// Remember that else only happens when the other if-statements were false.
else {
    // This will only happen when there isn't an enemy closer than 5 meters and cleave isn't ready.
}
```

Maniac Munchkins Solution

```
// Another chest in the field for the hero to break open!
// Attack the chest to break it open.
// Some munchkins won't stand idly by while you attack it!
// Defend yourself when a munchkin gets too close.

while(true) {
    var enemy = hero.findNearestEnemy();
    var distance = hero.distanceTo(enemy);
    if(hero.isReady('cleave')) {
        // First priority is to cleave if it's ready:
        hero.cleave();
    } else if(distance < 5) {
        // Attack the nearest munchkin that gets too close:
        hero.attack(enemy);
    } else {
        // Otherwise, try to break open the chest:
        hero.attack("Chest");
    }
}
```

#20. Forest Fire Dancing

Level Overview and Solutions

Intro

You can nest an `if-statement` inside another `if-statement` to make more complex choices.

Use this technique to find the safe spot and avoid the fireballs!

Default Code

```
// In this level the evilstone is bad! Avoid them walking the other direction.
while (true) {
    var evilstone = hero.findNearestItem();
    if (evilstone) {
        var pos = evilstone.pos;
        if (pos.x == 34) {
            // If the evilstone is on the left, go to the right side.

        } else {
            // If the evilstone is on the right, go to the left side.

        }
    } else {
        // If there's no evilstone, go to the middle.

    }
}
```

Overview

This situation might remind you of a previous level, Fire Dancing, but in this level, you have to dodge *two* fireballs at a time! On top of that, the fireballs are randomized, so you can't just go left and right in a loop. The logic for deciding where you can safely go is a little tricky:

- If you see a gem on one side of the map, move *away* from it to the other side;
- If there's no gem, move to the center.

Nested If-Statements

In order to make your strategy work, you have to use *nested* `if-statements`. That's where you put an `if-statement` *inside* another `if-statement` to make more complex choices inside choices. (Yo dawg...)

```
if (gem) {
    if (gem.pos.x == 34) {
        hero.say('left!');
    } else {
        hero.say('right!');
    }
} else {
    hero.say("middle!");
}
```

Note that the nested `if-statement` has extra indentation to show that it's *inside* the first one. The extra indentation is our way of showing that the left and right branches are in the inner `if-statement`, while the middle branch is part of the outer `if-statement`.

Positions

Each item object (and each unit) has a `pos` property, which stands for its position. And each `pos` is itself an object, which has `x` and `y` properties that you can use with `moveXY` and `buildXY`.

```
var enemy = hero.findNearestEnemy();
if (enemy) {
    var p = enemy.pos;
    hero.say(p.x);
    hero.say(p.y);
}
```

In the default code for this level, we set `pos` to `gem.pos` for you, but in future levels, you'll be seeing a lot of this property.

Forest Fire Dancing Solution

```
// In this level the evilstone is bad! Avoid them walking the other direction.
while (true) {
    var evilstone = hero.findNearestItem();
    if (evilstone) {
        var pos = evilstone.pos;
        if (pos.x == 34) {
            // If the evilstone is on the left, go to the right side.
            hero.moveXY(46, 22)
        } else {
            // If the evilstone is on the right, go to the left side.
            hero.moveXY(34, 22)
        }
    } else {
        // If there's no evilstone, go to the middle.
        hero.moveXY(40, 22)
    }
}
```

#21. Stillness in Motion

Level Overview and Solutions

Intro

Imagine each `if/else` as a container. It is acceptable to put another `if/else` statement inside of another! See:

```
var enemy = hero.findNearestEnemy();
if(enemy) {
    if(enemy.type == "munchkin") {
        hero.say("I see a munchkin!");
    } else {
        hero.say("I see some other kind of enemy!");
    }
} else {
    hero.say("I don't see any enemies...");
}
```

Remember to pay close attention to your `indentation`.

Default Code

```
// You can put one if-statement within another if-statement.
// Be careful how the if statements interact with each other.
// It's helpful to start with one outer if/else,
// using comments as placeholders for the inner if/else:

while(true) {
    var enemy = hero.findNearestEnemy();
    // If there is an enemy, then...
    if(enemy) {
        // Create a distance variable with distanceTo.

        // If the distance is less than 5 meters, then attack.

        // Else (the enemy is far away), then shield.

    // Else (there is no enemy)...
    } else {
        // ... then move back to the X.
        hero.moveXY(40, 34);
    }
}
```

Overview

For this level, you want to stay in the middle where the Headhunters can't see you!

You will use **nested-if-statements**.

When dealing with nested if statements, you need to pay close attention to how you set up the flow of your program.

If your if statements are complicated, it's often easier to build them up one at a time, using comments to fill in the future statements. For example, in this level we could begin by writing the following:

```
// If there is an enemy, then...
    // Do something
// Otherwise (there is no enemy)...
    // Move back to the X
```

Next, fill in the outer if/else statements, and the move, for real:

```
if(enemy) {
    // Do something
 } else {
    hero.moveXY(40, 34)
}
```

Now, let's detail the "Do something" :

```
if(enemy) {
    // If the enemy is less than 5 meters away, then attack
    // Otherwise (the enemy is far away), shield()
 } else {
    hero.moveXY(40, 34)
}
```

Finally, fill in the actual code for the inner if/else, making sure everything is indented correctly:

```
if(enemy) {
    if(hero.distanceTo(enemy) < 5) {
        hero.attack(enemy);
    } else {
        hero.shield();
    }
 } else {
    hero.moveXY(40, 34)
}
```

And, this entire block of if-s and else-s has to be indented under the **while-true loop** like:

```
while(true) {
    var enemy = hero.findNearestEnemy();

    if(enemy) {
        if(hero.distanceTo(enemy) < 5) {
            hero.attack(enemy);
        } else {
            hero.shield();
        }
     } else {
        hero.moveXY(40, 34)
    }

}
```

Hint: You can highlight several lines of code and press *Tab* to indent all of those lines, or *Shift+Tab* to un-indent all of those lines!

Stillness in Motion Solution

```
// You can put one if-statement within another if-statement.
// Be careful how the if statements interact with each other.
// It's helpful to start with one outer if/else,
// using comments as placeholders for the inner if/else:

while(true) {
    var enemy = hero.findNearestEnemy();
    // If there is an enemy, then...
    if(enemy) {
        // Create a distance variable with distanceTo.
        var distance = hero.distanceTo(enemy);
        // If the distance is less than 5 meters, then attack.
        if(distance < 5) {
            hero.attack(enemy);
        }
        // Else (the enemy is far away), then shield.
        else {
            hero.shield();
        }
    }
    // Else (there is no enemy)...
    else {
        // ... then move back to the X.
        hero.moveXY(40, 34);
    }
}
```

#22. The Agrippa Defense

Level Overview and Solutions

Intro

Don't waste your `cleave()` on only a few enemies. Be smart with `distanceTo()` to catch as many as possible.

Default Code

```
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Find the distance to the enemy with distanceTo.

        // If the distance is less than 5 meters...

            // ... if "cleave" is ready, cleave!

            // ... else, just attack.

    }
}
```

Overview

Sometimes it's best not to open with your strongest attack immediately. If you cleave at the first sight of the enemy, you may only catch the first few, leaving their friends to finish you off!

Try using `distanceTo()` to wait until the enemy is closer before you cleave. You can experiment to find the best range at which to strike; in this level, around **5 meters** works well.

Hint: If your cleave isn't ready, don't just stand there! Use a normal `attack()` while you wait for it to be ready again.

The Agrippa Defense Solution

```
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Find the distance to the enemy with distanceTo.
        var distance = hero.distanceTo(enemy);
        // If the distance is less than 5 meters...
        if(distance < 5) {
            // ... if "cleave" is ready, cleave!
            if(hero.isReady("cleave")) {
                hero.cleave(enemy);
            }
            // ... else, just attack.
            else {
                hero.attack(enemy);
            }
        }
    }
}
```

#22a. The Agrippa Defense A

Level Overview and Solutions

Intro

Don't waste your `cleave()` on only a few enemies. Be smart with `distanceTo()` to catch as many as possible.

Default Code

```
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Find the distance to the enemy with distanceTo.

        // If the distance is less than 5 meters...

            // ... if "cleave" is ready, cleave!

            // ... else, just attack.

    }
}
```

Overview

Sometimes it's best not to open with your strongest attack immediately. If you cleave at the first sight of the enemy, you may only catch the first few, leaving their friends to finish you off!

Try using `distanceTo()` to wait until the enemy is closer before you cleave. You can experiment to find the best range at which to strike; in this level, around **5 meters** works well.

Hint: If your cleave isn't ready, don't just stand there! Use a normal `attack()` while you wait for it to be ready again.

The Agrippa Defense A Solution

```
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Find the distance to the enemy with distanceTo.
        var distance = hero.distanceTo(enemy);
        // If the distance is less than 5 meters...
        if(distance < 5) {
            // ... if "cleave" is ready, cleave!
            if(hero.isReady("cleave")) {
                hero.cleave(enemy);
            }
            // ... else, just attack.
            else {
                hero.attack(enemy);
            }
        }
    }
}
```

#22b. The Agrippa Defense B

Level Overview and Solutions

Intro

Don't waste your `cleave()` on only a few enemies. Be smart with `distanceTo()` to catch as many as possible.

Default Code

```
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Find the distance to the enemy with distanceTo.

        // If the distance is less than 5 meters...

            // ... if "cleave" is ready, cleave!

            // ... else, just attack.

    }
}
```

Overview

Sometimes it's best not to open with your strongest attack immediately. If you cleave at the first sight of the enemy, you may only catch the first few, leaving their friends to finish you off!

Try using `distanceTo()` to wait until the enemy is closer before you cleave. You can experiment to find the best range at which to strike; in this level, around **5 meters** works well.

Hint: If your cleave isn't ready, don't just stand there! Use a normal `attack()` while you wait for it to be ready again.

The Agrippa Defense B Solution

```
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Find the distance to the enemy with distanceTo.
        var distance = hero.distanceTo(enemy);
        // If the distance is less than 5 meters...
        if(distance < 5) {
            // ... if "cleave" is ready, cleave!
            if(hero.isReady("cleave")) {
                hero.cleave(enemy);
            }
            // ... else, just attack.
            else {
                hero.attack(enemy);
            }
        }
    }
}
```

#23. Village Rover

Level Overview and Solutions

Intro

Patrolling the village can be boring and repetetive.

Using `functions` makes repeating the same task easier. You can turn many lines of code into just one!

```
// Define a function
function findAndAttackEnemy() {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
    }
};

// Now just one line can be used in place of all that code
findAndAttackEnemy();
```

Default Code

```
// This defines a function called findAndAttackEnemy
function findAndAttackEnemy() {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
    }
}

// This code is not part of the function.
while(true) {
    // Now you can patrol the village using findAndAttackEnemy
    hero.moveXY(35, 34);
    findAndAttackEnemy();

    // Now move to the right entrance.

    // Use findAndAttackEnemy

}
```

Overview

Functions are an important part of coding.

You've been using functions all along: any time you write code like:

```
hero.attack(enemy);
```

...you are "calling" (or "invoking") a function called `attack`.

The actual code that gets executed when you call `attack` is long and complex. Imagine if you had to write 25 lines of code in your program each time you wanted to swing your sword!

That's the first benefit of functions: they reduce a whole bunch of code down into one line.

Not only does this save you from having to re-type the same code over and over, it also makes your code easier to understand, because it takes what might really be complicated logic ("Ok so I want to attack. Do I have a weapon? Am I close enough to hit with my weapon? How long does it take to use my weapon? Do I hit? Do I cause damage?"), and makes it an easy to understand idea: `attack`.

Now you will not only be calling functions, you will **define** your own functions!

Defining a function has two parts: the **name** and the **body**.

The name is the thing you will use to call the function later, like `attack`.

The body is the code that will be executed when the function is called.

There are also sometimes **arguments** (like the *enemy* in `attack(enemy)`) but we will get into that in future levels.

Village Rover Solution

```
// This defines a function called findAndAttackEnemy
function findAndAttackEnemy() {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
    }
}

// This code is not part of the function.
while(true) {
    // Now you can patrol the village using findAndAttackEnemy
    hero.moveXY(35, 34);
    findAndAttackEnemy();

    // Now move to the right entrance.
    hero.moveXY(60, 31);
    // Use findAndAttackEnemy
    findAndAttackEnemy();
}
```

#24. Village Warder

Level Overview and Solutions

Intro

The village is under attack by an even bigger horde of Ogres! You'll need to use functions to attack *and* to cleave against this onslaught.

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

Default Code

```
// This function attacks the nearest enemy.
function findAndAttackEnemy() {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
    }
}

// Define a function to cleave enemies (but only when the ability is ready).
function findAndCleaveEnemy() {
    // Find the nearest enemy:

    // If an enemy exists:

        // And if "cleave" is ready:

            // It's time to cleave!

}

// In your main loop, patrol, cleave, and attack.
while (true) {
    // Move to the patrol point, cleave, and attack.
    hero.moveXY(35, 34);
    findAndCleaveEnemy();
    findAndAttackEnemy();

    // Move to the other point:

    // Use findAndCleaveEnemy function:

    // Use findAndAttackEnemy function:

}
```

Overview

In this level you'll be writing the *definition* of a function so that you can *call* it in your main loop. Because the Ogres might attack in larger waves, you'll need to add a function that uses your Cleave ability.

When you define your `findAndCleaveEnemy` function, remember to check that: 1. the enemy exists; and 2. that your Cleave ability is ready to use.

Be sure to remember how to define a function:

```
function sayHello() {
    hero.say("Hello!");
}
```

Remember that when you call a function you *don't* add `hero` to it, because the function is defined by *you*, not the hero.

```
sayHello();
hero.say("Goodbye.");
```

Village Warder Solution

```
// This function attacks the nearest enemy.
function findAndAttackEnemy() {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        hero.attack(enemy);
    }
}

function findAndCleaveEnemy() {
    // Define a function to cleave enemies (but only when the ability is ready).
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        if (hero.isReady("cleave")) {
            hero.cleave(enemy);
        }
    }
}

// In your main loop, patrol, cleave, and attack.
while (true) {
    // Move to the patrol point, cleave, and attack.
    hero.moveXY(35, 34);
    findAndCleaveEnemy();
    findAndAttackEnemy();

    // Move to the other point:
    hero.moveXY(60, 31);
    findAndCleaveEnemy();
    findAndAttackEnemy();
}
```

#25. Village Champion

Level Overview and Solutions

Intro

The ogres have broken through and opened another path into the village! You'll need to write your own function to handle this new attack.

Look closely at how to define a function. You're on your own this time!

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

Default Code

```
// Incoming munchkins! Defend the town!

// Define your own function to fight the enemy!
function cleaveOrAttack() {
    // In the function, find an enemy, then cleave or attack it.

}


// Move between patrol points and call the function.
while (true) {
    hero.moveXY(35, 34);
    // Use cleaveOrAttack function you defined above.

    hero.moveXY(47, 27);
    // Call the function again.

    hero.moveXY(60, 31);
    // Call the function again.

}
```

Overview

In this level you need to define your own function to fight off the ogres. This function should combine the cleaving and attacking logic from "Village Warder" into one. Go back and look at that level to remind yourself how to define a function.

The code for cleaving and attacking requires some nested `if/else` statements. Go back and look at "Stillness in Motion" and "The Agrippa Defense" to remind yourself how to structure your logic.

Remember how to define a function:

```
function sayHello() {
    hero.say("Hello!");
}
```

Also, remember that when you call a function you *don't* add `hero` to it, because the function is defined by *you*, not the hero.

```
sayHello();
hero.say("Goodbye.");
```

Village Champion Solution

```
// Define your own function to fight the enemy!
// In the function, find an enemy, then cleave or attack it.
function cleaveOrAttack() {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        if (hero.isReady("cleave")) {
            hero.cleave(enemy);
        } else {
            hero.attack(enemy);
        }
    }
}

// Move between patrol points and call the function.
while (true) {
    hero.moveXY(35, 34);
    // Use cleaveOrAttack function you defined above.
    cleaveOrAttack();

    hero.moveXY(47, 27);
    // Call the function again.
    cleaveOrAttack();

    hero.moveXY(60, 31);
    // Call the function again.
    cleaveOrAttack();
}
```

#26. A Fine Mint

Level Overview and Solutions

Intro

Now you'll need to write your own function! We've given you `pickUpCoin` as an example, but you need to create `attackEnemy`:

```
function attackEnemy() {
    // put your attack code here
}
```

Default Code

```
// Peons are trying to steal your coins!
// Write a function to squash them before they can take your coins.

function pickUpCoin() {
    var coin = hero.findNearestItem();
    if(coin) {
        hero.moveXY(coin.pos.x, coin.pos.y);
    }
}

// Write the attackEnemy function below.
// Find the nearest enemy and attack them if they exist!


while(true) {
    //attackEnemy(); // Δ Uncomment this line after you write an attackEnemy function.
    pickUpCoin();
}
```

Overview

In this level you'll be writing a function from scratch. Be sure to remember how to define a function:

```
function sayHello() {
    hero.say("Hello!");
}
```

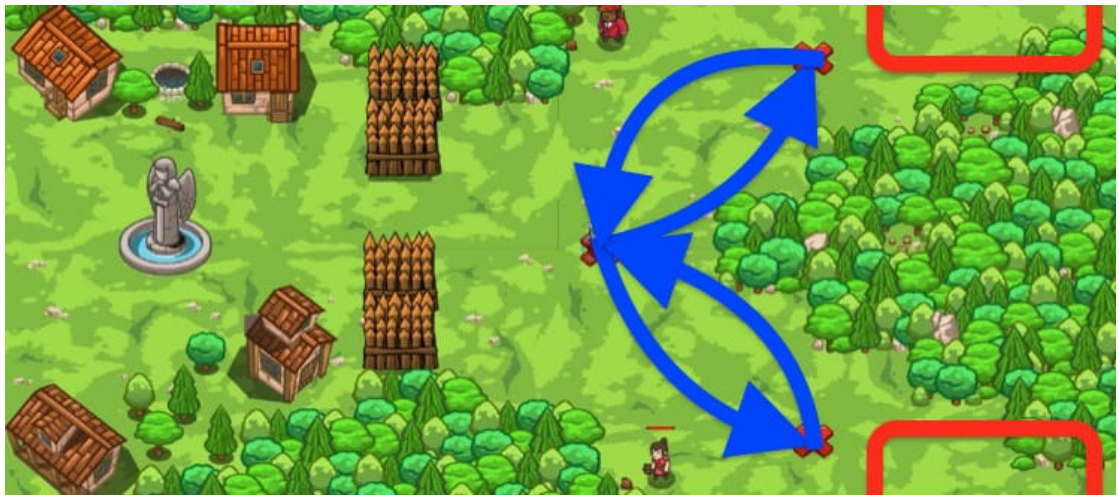
A Fine Mint Solution

```
// Peons are trying to steal your coins!
// Write a function to squash them before they can take your coins.

function pickUpCoin() {
    var coin = hero.findNearestItem();
    if(coin) {
        hero.moveXY(coin.pos.x, coin.pos.y);
    }
}

// Write the attackEnemy function below.
// Find the nearest enemy and attack them if they exist!
function attackEnemy() {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        hero.attack(enemy);
    }
}

while(true) {
    attackEnemy(); // Δ Uncomment this line after you write an attackEnemy function.
    pickUpCoin();
}
```

#27. Backwoods Fork

Level Overview and Solutions

Intro



A `function` is an important concept when writing code! They are used to separate individual, **repeatable** parts of your code into easier to digest pieces.

Often times, a function requires an `argument`. This is a way to customize a **repeatable** action while still optimizing and cutting down the length of one's code. This is what you put in between `(` and `)` when `calling` the function.

A `parameter` is what a potential `argument` is called inside the function definition.

```
// This function has 1 parameter: 'target':
function checkAndAttack(target) {
    // 'target' is a predefined variable.
    if(target) {
        hero.attack(target);
    }
}
var enemy = hero.findNearestEnemy();
// Below, 'enemy' is the argument. This becomes 'target' inside checkAndAttack.
checkAndAttack(enemy);
```

Default Code

```
// Incoming ogres!
// Use the checkAndAttack function to make your code easy to read.

// This function has a parameter.
// An parameter is a way of passing information into a function.
function checkAndAttack(target) {
    // The 'target' parameter is just a variable!
    // It contains the argument when the function was called.
    if(target) {
        hero.attack(target);
    }
    hero.moveXY(43, 34);
}

while(true) {
    hero.moveXY(58, 52);
    var topEnemy = hero.findNearestEnemy();
    checkAndAttack(topEnemy);

    // Move to the bottom X mark.

    // Create a variable named bottomEnemy and find the nearest enemy.

    // Use the checkAndAttack function, and include the bottomEnemy variable.

}
```

Overview

`Functions` are an important part of writing code! They can be used to separate individual parts of code to help understand each bit easier.

`Functions` can do much more than just do some actions independent of any input! Using `arguments` and `parameters`, `functions` are capable of acting on what is sent in.

An `argument` is the information included between the `()` when any given `function` is called.

A `parameter` is a variable containing the information passed in from a `function`. It is just a predefined variable containing any information sent in!

```
// 'target' after the function name is called a 'parameter'.
// Think of a parameter as a new variable containing information from outside the function!
function checkAndAttack(target) {
    // 'target' is a predefined variable by the parameter, so nothing more needed to use it!
    if(target) { // Check if the 'target' exists.
        hero.attack(target); // If so, attack!
    }
}
var enemy = hero.findNearestEnemy();
// Below, the variable 'enemy' is an 'argument'.
// Including 'arguments' when calling functions lets them behave differently depending on the input!
// In this case, we'll attack whatever enemy is passed in!
hero.checkAndAttack(enemy);
```

Backwoods Fork Solution

```
// Incoming ogres!
// Use the checkAndAttack function to make your code easy to read.

// This function has a parameter.
// An parameter is a way of passing information into a function.
function checkAndAttack(target) {
    // The 'target' parameter is just a variable!
    // It contains the argument when the function was called.
    if(target) {
        hero.attack(target);
    }
    hero.moveXY(43, 34);
}

while(true) {
    hero.moveXY(58, 52);
    var topEnemy = hero.findNearestEnemy();
    checkAndAttack(topEnemy);

    // Move to the bottom X mark.
    hero.moveXY(58, 16);
    // Create a variable named bottomEnemy and find the nearest enemy.
    var bottomEnemy = hero.findNearestEnemy();
    // Use the checkAndAttack function, and include the bottomEnemy variable.
    checkAndAttack(bottomEnemy);
}
```

#28. Tomb Raider

Level Overview and Solutions

Intro

Remember that a `parameter` is a way of passing information into a function. It is a predefined variable of whatever the argument is when called!

```
function checkAndEat(target) {
    if(target.type == "fruit") {
        hero.eat(target);
    } else {
        hero.toss(target)
    }
}
while(true) {
    hero.moveUp()
    var nearestTree = hero.findNearestTree()
    var food = hero.harvest(nearestTree)
    checkAndEat(food)
}
```

Default Code

```
// A forgotten tomb in the forest!
// But the ogres are hot on your heels.
// Break open the tomb, while defending yourself from the munchkins.

// This function should attack an enemy if it exists, otherwise attack the door!
function checkToDefend(target) {
    // Check if the target exists

        // If so, attack the target

    // Use an else to do something if there is no target

        // Otherwise attack the "Door"

}

while(true) {
    var enemy = hero.findNearestEnemy();
    checkToDefend(enemy);
}
```

Overview

It's time to fill in the body of a function and use it to make your code nice and clean!

Remember that a `parameter` is just a way of passing information into a standalone function. It is a predefined variable containing whatever was when between the `()` when the function was `called`.

'Calling' a function is when the code is actually peformed. See the examples below as to how functions are called:

```
// This is 'defining' a function:
function moveUpAndDown() {
    hero.moveUp(); // This is calling the 'moveUp' function.
    hero.moveDown(); // This is calling the 'moveDown' function.
}
hero.say("I'm saying something!"); // This is 'calling' the 'say' method.
moveUpAndDown(); // This is 'calling' the custom defined 'moveUpAndDown' method.
```

Just use the target like you would any variable!

```
function checkAndDefend(target) {
    if(target) {
        hero.say("I see an enemy! I should beat them up.");
    }
}
```

Tomb Raider Solution

```
// A forgotten tomb in the forest!
// But the ogres are hot on your heels.
// Break open the tomb, while defending yourself from the munchkins.

// This function should attack an enemy if it exists, otherwise attack the door!
function checkToDefend(target) {
    // Check if the target exists
    if(target) {
        // If so, attack the target
        hero.attack(target);
    }
    // Use an else to do something if there is no target
    else {
        // Otherwise attack the "Door"
        hero.attack("Door");
    }
}

while(true) {
    var enemy = hero.findNearestEnemy();
    checkToDefend(enemy);
}
```

#29. Tomb Ghost

Level Overview and Solutions

Intro

Practice using a `parameter` passed into a `function`:

```
function hitAndRun(target) {
    if(target) {
        hero.attack(target);
        hero.moveXY(10, 20);
    }
}

var enemy = hero.findNearestEnemy();
hitAndRun(enemy); // Call hitAndRun with target set to enemy
```

Default Code

```
// The only exit is blocked by ogres.
// Hide from the skeletons and defeat the ogres one by one.

// This function should attack an enemy and hide.
function hitOrHide(target) {
    // If 'target' exists:

        // Attack 'target'.

        // Then move to the red mark.

}

while (true) {
    var enemy = hero.findNearestEnemy();
    hitOrHide(enemy);
}
```

Overview

Remember that a **parameter** is a way of passing information into a function. It is a predefined variable containing whatever was put between the `()` when the function was `called`.

Just use the `target` parameter like you would any variable:

```
function checkAndDefend(target) {
    if(target) {
        hero.say("I see an enemy! I should defeat them!");
    }
}
```

Tomb Ghost Solution

```
// The only exit is blocked by ogres.
// Hide from the skeletons and defeat the ogres one by one.

// This function should attack an enemy and hide.
function hitOrHide(target) {
    // If 'target' exists:
    if (target) {
        // Attack 'target'.
        hero.attack(target);
        // Then move to the red mark.
        hero.moveXY(32, 17);
    }
}

while (true) {
    var enemy = hero.findNearestEnemy();
    hitOrHide(enemy);
}
```

#30. Seek-and-Hide

Level Overview and Solutions

Intro

Your goal is to move to each of the red X marks, searching for lightstones. If you find one, you'll have to hide at the center X mark before continuing your search.

First, finish the `checkTakeHide()` function to hide in the center of the camp after finding a lightstone.

Then, examine the sample code that calls `checkTakeHide(stone)` at the right X mark, and then write your own code to call it at the left X mark.

Default Code

```
// Gather 4 lightstones to defeat the Brawler.
// If you find a lightstone, hide.

function checkTakeHide(item) {
    if (item) {
        // The item is here, so take it.
        hero.moveXY(item.pos.x, item.pos.y);
        // Then move to the center of the camp (40, 34)

    }
}

while (true) {
    // Move to the top right X mark.
    hero.moveXY(68, 56);
    // Search for a lightstone there.
    var lightstone = hero.findNearestItem();
    // Call checkTakeHide with the argument: lightstone
    checkTakeHide(lightstone);

    // Move to the top left mark.

    // Search for a lightstone.

    // Call the checkTakeHide function.
    // Pass in the result of your search as an argument.

}
```

Overview

You can use a function parameter as a variable inside the function. But also you can add additional instructions, which are not related to the parameter. For example:

```
function checkAndHit(unit) {
    if (unit) {
        hero.attack(unit);
        // An additional instruction without 'unit'.
        hero.say("I'm dangerous!");
    }
}
```

Also, don't forget you can call the same function as many times as you want.

```
hero.moveXY(10, 10);
var enemy = hero.findNearestEnemy();
checkAndHit(enemy);
// Next point.
hero.moveXY(70, 10);
enemy = hero.findNearestEnemy();
checkAndHit(enemy);
```

Seek-and-Hide Solution

```
// Gather 4 lightstones to defeat the Brawler.
// If you find a lightstone, hide.

function checkTakeHide(item) {
    if (item) {
        // The item is here, so take it.
        hero.moveXY(item.pos.x, item.pos.y);
        // Then move to the center of the camp (40, 34)
        hero.moveXY(40, 34);
    }
}

while (true) {
    // Move to the top right X mark.
    hero.moveXY(68, 56);
    // Search for a lightstone there.
    var lightstone = hero.findNearestItem();
    // Call checkTakeHide with the argument: lightstone
    checkTakeHide(lightstone);

    // Move to the top left mark.
    hero.moveXY(12, 56);
    // Search for a lightstone.
    lightstone = hero.findNearestItem();
    // Call the checkTakeHide function.
    // Pass in the result of your search as an argument.
    checkTakeHide(lightstone);
}
```

#31. Forest Miners

Level Overview and Solutions

Intro

It's your job to make sure no ogres disturb these miners.

First, fill in the `checkEnemyOrSafe()` function. Only call the peasants if there were no ogres, because attacking an ogre will attract attention!

Then, look at the code that checks the top-right X mark, and do something similar to check the bottom-left X mark.

Default Code

```
// Check if the mines are safe for the workers.

function checkEnemyOrSafe(target) {
    // If target (the parameter) exists:

        // Then attack target.

    // Otherwise:

        // Use say() to call the peasants.

}

while (true) {
    // Move to, and check the top right X mark.
    hero.moveXY(64, 54);
    var enemy1 = hero.findNearestEnemy();
    checkEnemyOrSafe(enemy1);

    // Move to the bottom left X mark.

    // Save the result of findNearestEnemy() in a variable.
    var enemy2 = hero.findNearestEnemy();
    // Call checkEnemyOrSafe, and pass the
    // result of findNearestEnemy as the argument.

}
```

Overview

You've learned how to write and call functions with parameters in the previous levels. If you have problems with this level, you can return and repeat levels about functions to refresh your skills.

Forest Miners Solution

```
// Check if the mines are safe for the workers.

function checkEnemyOrSafe(target) {
    // If target (the parameter) exists:
    if (target) {
        // Then attack target.
        hero.attack(target);
    }
    // Otherwise:
    else {
        // Use say() to call the peasants.
        hero.say("All clear!");
    }
}

while (true) {
    // Move to, and check the top right X mark.
    hero.moveXY(64, 54);
    var enemy1 = hero.findNearestEnemy();
    checkEnemyOrSafe(enemy1);

    // Move to the bottom left X mark.
    hero.moveXY(16, 14);
    // Save the result of findNearestEnemy() in a variable.
    var enemy2 = hero.findNearestEnemy();

    // Call checkEnemyOrSafe, and pass the
    // result of findNearestEnemy as the argument.
    checkEnemyOrSafe(enemy2);
}
```

#32. Leave it to Cleaver

Level Overview and Solutions

Intro

The function `cleaveWhenClose` defines a parameter called `target`:

```
function cleaveWhenClose(target) {
    if(hero.distanceTo(target) < 5) {
        // cleave or attack here
    }
}
```

This allows you to pass the enemy as an argument when calling the function:

```
cleaveWhenClose(enemy)
```

Default Code

```
// This shows how to define a function called cleaveWhenClose
// The function defines a parameter called target
function cleaveWhenClose(target) {
    if(hero.distanceTo(target) < 5) {
        // Put your attack code here
        // If cleave is ready, then cleave target

        // else, just attack target!

    }
}

// This code is not part of the function.
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Note that inside cleaveWhenClose, we refer to the enemy as target.
        cleaveWhenClose(enemy);
    }
}
```

Overview

In previous levels, you've passed *arguments* to functions. When you do `hero.say("Hello!")`, you are passing the String `"Hello!"` as an argument to the function `say`.

Now, you'll learn to define *parameters*, which is what *arguments* are called when defining our own functions.

For this level, you define a function called `cleaveWhenClose` that accepts a parameter called `target`:

```
function cleaveWhenClose(target) {
    if(hero.distanceTo(target) < 5) {
        // cleave or attack here
    }
}
```

Note that when you call the function later, it says:

```
cleaveWhenClose(enemy)
```

`enemy` is what the enemy is called outside the function, `target` is what the enemy is called inside the function. This is two different variables that both point to the same ogre!

Leave it to Cleaver Solution

```
// This shows how to define a function called cleaveWhenClose
// The function defines a parameter called target
function cleaveWhenClose(target) {
    if(hero.distanceTo(target) < 5) {
        // Put your attack code here
        // If cleave is ready, then cleave target
        if(hero.isReady("cleave")) {
            hero.cleave();
        }
        // else, just attack target!
        else {
            hero.attack(target);
        }
    }
}

// This code is not part of the function.
while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Note that inside cleaveWhenClose, we refer to the enemy as target.
        cleaveWhenClose(enemy);
    }
}
```

#33. Return to Thornbush Farm

Level Overview and Solutions

Intro

Your functions can define more than one parameter.

```
function maybeBuildTrap(x, y) {
    // When this function is called below,
    // x will be 43, y will be 50
}

maybeBuildTrap(43, 50)
```

Default Code

```
// The function maybeBuildTrap defines TWO parameters!
function maybeBuildTrap(x, y) {
    // Use x and y as the coordinates to move to.
    hero.moveXY(x, y);
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Use buildXY to build a "fire-trap" at the given x and y.

    }
}

while(true) {
    // This calls maybeBuildTrap, with the coordinates of the top entrance.
    maybeBuildTrap(43, 50);

    // Now use maybeBuildTrap at the left entrance!

    // Now use maybeBuildTrap at the bottom entrance!

}
```

Overview

Just like `moveXY` accepts two arguments, the functions you create can define more than one parameter!

```
function maybeBuildTrap(x, y) {
    // When this function is called below,
    // x will be 43, y will be 50
}

maybeBuildTrap(43, 50)
```

Parameters vs. Arguments

So why do we sometimes call things parameters, and sometimes call things arguments?

A *parameter* is the thing you define in your function.

An *argument* is the actual value that is passed into the function when it's called by your program!

Return to Thornbush Farm Solution

```
// The function maybeBuildTrap defines TWO parameters!
function maybeBuildTrap(x, y) {
    // Use x and y as the coordinates to move to.
    hero.moveXY(x, y);
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Use buildXY to build a "fire-trap" at the given x and y.
        hero.buildXY("fire-trap", x, y);
    }
}

while(true) {
    // This calls maybeBuildTrap, with the coordinates of the top entrance.
    maybeBuildTrap(43, 50);

    // Now use maybeBuildTrap at the left entrance!
    maybeBuildTrap(25, 34);

    // Now use maybeBuildTrap at the bottom entrance!
    maybeBuildTrap(43, 20);

}
```

#33a. Return to Thornbush Farm A

Level Overview and Solutions

Intro

Your functions can define more than one parameter.

```
function maybeBuildTrap(x, y) {
    // When this function is called below,
    // x will be 43, y will be 50
}

maybeBuildTrap(43, 50)
```

Default Code

```
// The function maybeBuildTrap defines TWO parameters!
function maybeBuildTrap(x, y) {
    // Use x and y as the coordinates to move to.
    hero.moveXY(x, y);
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Use buildXY to build a "fire-trap" at the given x and y.

    }
}

while(true) {
    // This calls maybeBuildTrap, with the coordinates of the left entrance.
    maybeBuildTrap(20, 34);

    // Now use maybeBuildTrap at the bottom entrance!

    // Now use maybeBuildTrap at the right entrance!

}
```

Overview

Just like `moveXY` accepts two arguments, the functions you create can define more than one parameter!

```
function maybeBuildTrap(x, y) {
    // When this function is called below,
    // x will be 43, y will be 50
}

maybeBuildTrap(43, 50)
```

Parameters vs. Arguments

So why do we sometimes call things parameters, and sometimes call things arguments?

A *parameter* is the thing you define in your function.

An *argument* is the actual value that is passed into the function when it's called by your program!

Return to Thornbush Farm A Solution

```
// The function maybeBuildTrap defines TWO parameters!
function maybeBuildTrap(x, y) {
    // Use x and y as the coordinates to move to.
    hero.moveXY(x, y);
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Use buildXY to build a "fire-trap" at the given x and y.
        hero.buildXY("fire-trap", x, y);
    }
}

while(true) {
    // This calls maybeBuildTrap, with the coordinates of the left entrance.
    maybeBuildTrap(20, 34);

    // Now use maybeBuildTrap at the bottom entrance!
    maybeBuildTrap(38, 20);

    // Now use maybeBuildTrap at the right entrance!
    maybeBuildTrap(56, 34);

}
```

#33b. Return to Thornbush Farm B

Level Overview and Solutions

Intro

Your functions can define more than one parameter.

```
function maybeBuildTrap(x, y) {
    // When this function is called below,
    // x will be 43, y will be 50
}

maybeBuildTrap(43, 50)
```

Default Code

```
// The function maybeBuildTrap defines TWO parameters!
function maybeBuildTrap(x, y) {
    // Use x and y as the coordinates to move to.
    hero.moveXY(x, y);
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Use buildXY to build a "fire-trap" at the given x and y.

    }
}

while(true) {
    // This calls maybeBuildTrap, with the coordinates of the bottom entrance.
    maybeBuildTrap(38, 20);

    // Now use maybeBuildTrap at the right entrance!

    // Now use maybeBuildTrap at the top entrance!

}
```

Overview

Just like `moveXY` accepts two arguments, the functions you create can define more than one parameter!

```
function maybeBuildTrap(x, y) {
    // When this function is called below,
    // x will be 43, y will be 50
}

maybeBuildTrap(43, 50)
```

Parameters vs. Arguments

So why do we sometimes call things parameters, and sometimes call things arguments?

A *parameter* is the thing you define in your function.

An *argument* is the actual value that is passed into the function when it's called by your program!

Return to Thornbush Farm B Solution

```
// The function maybeBuildTrap defines TWO parameters!
function maybeBuildTrap(x, y) {
    // Use x and y as the coordinates to move to.
    hero.moveXY(x, y);
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        // Use buildXY to build a "fire-trap" at the given x and y.
        hero.buildXY("fire-trap", x, y);
    }
}

while(true) {
    // This calls maybeBuildTrap, with the coordinates of the bottom entrance.
    maybeBuildTrap(38, 20);

    // Now use maybeBuildTrap at the right entrance!
    maybeBuildTrap(56, 34);

    // Now use maybeBuildTrap at the top entrance!
    maybeBuildTrap(38, 48);

}
```

#34. Agrippa Refactored

Level Overview and Solutions

Intro

Did you find "The Agrippa Defense" difficult? Functions can help you clean up your code and make it easier to read.

In this level, you'll replay the same scenario, but this time, you'll solve it much more easily using functions that take arguments! When you want a function to accept arguments, list them in the function's declaration:

```
function valentine(a, b) {
    hero.say(a + " loves " + b);
}

valentine("Angie", "Bobby");
valentine("Bobby", "Carla");
```

Default Code

```
function cleaveOrAttack(enemy) {
    // If "cleave" is ready, cleave; otherwise, attack.

}

while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        var distance = hero.distanceTo(enemy);
        if(distance < 5) {
            // Call the "cleaveOrAttack" function, defined above.
            cleaveOrAttack(enemy);
        }
    }
}
```

Overview

Back in "Village Rover," we used functions to avoid having to repeat the same code over and over. Here, we use functions to clean up our code and make it less complicated.

Your solution to "The Agrippa Defense" probably looked something like this:

```
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        var  = hero.distanceTo(enemy);
        if (distance < 5) {
            if (hero.isReady("cleave")) {
                hero.cleave(enemy);
            } else {
                hero.attack(enemy);
            }
        }
    }
}
```

That's a lot of nested `if` statements! To make it easier to manage all those `if`s, you can move the innermost block of code into a separate function that can be called in just one line:

```
while (true) {
    var enemy = hero.findNearestEnemy();
    if (enemy) {
        var distance = hero.distanceTo(enemy);
        if (distance < 5) {
            cleaveOrAttack(enemy);
        }
    }
}
```

Notice that when we call `cleaveOrAttack`, we pass the enemy to it as an argument. When you want a function to accept arguments like this, you need to declare them in the function's declaration:

```
function valentine(a, b) {
    hero.say(a + " loves " + b);
};

valentine("Angie", "Bobby");
valentine("Bobby", "Carla");
```

Agrippa Refactored Solution

```
function cleaveOrAttack(enemy) {
    // If "cleave" is ready, cleave; otherwise, attack.
    if(hero.isReady("cleave")) {
        hero.cleave(enemy);
    } else {
        hero.attack(enemy);
    }
}

while(true) {
    var enemy = hero.findNearestEnemy();
    if(enemy) {
        var distance = hero.distanceTo(enemy);
        if(distance < 5) {
            // Call the "cleaveOrAttack" function, defined above.
            cleaveOrAttack(enemy);
        }
    }
}
```

#35. Closed Crossroad

Level Overview and Solutions

Intro

The village is located on a busy crossroad and only a skilled builder can protect it! Move around the village clockwise.

Use `"fire-trap"` s for the top and the bottom passages. Use `"fence"` s for the left and the right passages.

Functions can define many parameters with various types such as strings or numbers.

```
function maybeBuildSomething(buildType, x, y) {
    // When this function is called below,
    // buildType will be "fence"
    // x will be 20 and y will be 40
}

maybeBuildSomething("fence", 20, 40);
```

Default Code

```
// Only build if you see an enemy.

// The function defines THREE parameters.
function maybeBuildSomething(buildType, x, y) {
    // Move to the position specified by x and y parameters.

    // Find the nearest enemy.

    // If there is an enemy

        // then use buildXY with buildType, x, and ythen use buildXY with buildType, x, and y

}

while(true) {
    // Call maybeBuildSomething with "fire-trap" and the coordinates of the bottom X.
    maybeBuildSomething("fire-trap", 40, 20);
    // Call maybeBuildSomething, with "fence" at the left X!
    maybeBuildSomething("fence", 26, 34);
    // Call maybeBuildSomething with "fire-trap" at the top X!
    maybeBuildSomething("fire-trap", 40, 50);
    // Call maybeBuildSomething with "fence" at the right X!
    maybeBuildSomething("fence", 54, 34);
}
```

Overview

Functions can have any number of parameters, and they can be different types.

Like `buildXY`, which has 3 parameters: a string (the type of object to build), and two numbers (the x and y coordinates).

```
function maybeBuildSomething(buildType, x, y) {
    // When this function is called below,
    // buildType will be 'fence', x will be 40, y will be 20
}

maybeBuildSomething('fence', 40, 20);
```

`buildType` is a string, `x` and `y` are numbers.

Note: When you call a function, the order of arguments should be the same as the order they're defined in.

Now you can use one function for `"fence"`s and `"fire-trap"`s instead of two and make your code simpler!

Closed Crossroad Solution

```
// Only build if you see an enemy.

// The function defines THREE parameters.
function maybeBuildSomething(buildType, x, y) {
    // Move to the position specified by x and y parameters.
    hero.moveXY(x, y);
    // Find the nearest enemy.
    var enemy = hero.findNearestEnemy();
    // If there is an enemy
    if (enemy) {
        // then use buildXY with buildType, x, and y
        hero.buildXY(buildType, x, y);
    }
}

while(true) {
    // Call maybeBuildSomething with "fire-trap" and the coordinates of the bottom X.
    maybeBuildSomething("fire-trap", 40, 20);
    // Call maybeBuildSomething, with "fence" at the left X!
    maybeBuildSomething("fence", 26, 34);
    // Call maybeBuildSomething with "fire-trap" at the top X!
    maybeBuildSomething("fire-trap", 40, 50);
    // Call maybeBuildSomething with "fence" at the right X!
    maybeBuildSomething("fence", 54, 34);
}
```

#36. Greed Traps

Level Overview and Solutions

Intro

Lure ogres into traps with coins!

Your task is to patrol (move to each X mark in turn) and watch for coins.

If a coin appears, then build a fire trap on the X mark.

Peasants are roaming the forest as well so **only build traps when you see a coin.**

Avoid collecting the coins because they act as bait for you traps.

Default Code

```
// Patrol and place traps ONLY if you see a coin.

// Write this function.
function maybeBuildTrap(x, y) {
    // Move to the given x,y postion

    // Search a coin and  if you see it, build a "fire-trap"

}

while (true) {
    // Call maybeBuildTrap for the top left passage.
    maybeBuildTrap(12, 56);
    // Now for the top right passage.
    maybeBuildTrap(68, 56);
    // Now for the bottom right passage.

    // Now for the bottom left passage.

}
```

Overview

The previous levels should have prepared you well for this challenge. The only difference is that you need to write the function yourself.

If you have problems with this level, then try to return previous levels about function and refresh your skills.

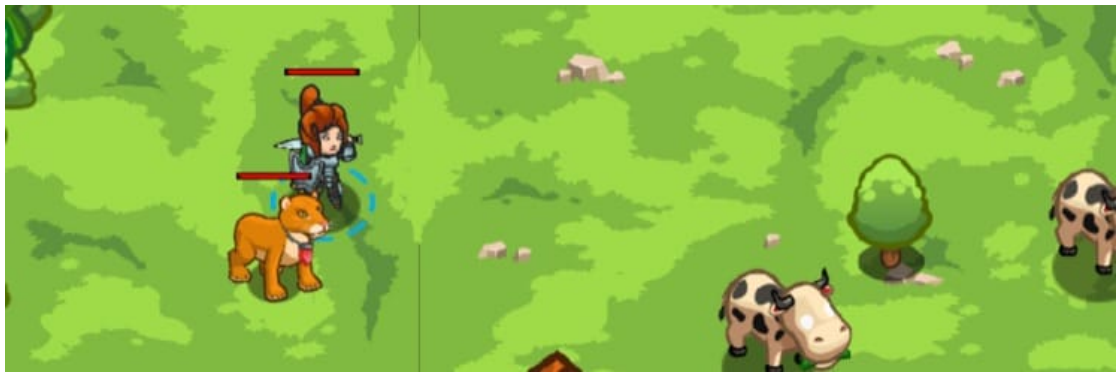
Greed Traps Solution

```
// Patrol and place traps ONLY if you see a coin.

// Write this function.
function maybeBuildTrap(x, y) {
    // Move to the given x,y postion
    hero.moveXY(x, y);
    // Search a coin and  if you see it, build a "fire-trap"
    var coin = hero.findNearestItem();
    if (coin) {
        hero.buildXY("fire-trap", x, y);
    }

}

while (true) {
    // Call maybeBuildTrap for the top left passage.
    maybeBuildTrap(12, 56);
    // Now for the top right passage.
    maybeBuildTrap(68, 56);
    // Now for the bottom right passage.
    maybeBuildTrap(68, 12);
    // Now for the bottom left passage.
    maybeBuildTrap(12, 12);
}
```

#37. Backwoods Buddy

Level Overview and Solutions

Intro



You have a pet!

Your pet responds to `events`. When she hears something, a `"hear"` event is triggered.

To have your pet respond to a `"hear"` event, you use a `handler function`:

```
function speak(event) {
    pet.say("Meow!");
}

pet.on("hear", speak);
```

Default Code

```
// You now have a pet!

function speak(event) {
    // Your pet should respond with pet.say()

}
// This tells your pet to run the speak() function when she hears something
pet.on("hear", speak);

// Talk to your pet!
hero.say("Hello Kitty");
```

Overview

This level introduces pets! Yay!

You refer to your pet using `pet`, just like you refer to your hero using `hero`.

You train your pet to do things by using **event handlers**.

An **event** is something that happens that your pet can react to, such as `"hear"`ing you speak!

An **event handler** is a function that will be executed when an **event** happens.

To train your pet to react to an **event**, use the `on()` function:

```
pet.on("hear", speak);
```

This means your pet will execute the function `speak()` when it `"hear"`s someone say something.

Backwoods Buddy Solution

```
// You now have a pet!

function speak(event) {
    // Your pet should respond with pet.say()
    pet.say("Meow!");
}
// This tells your pet to run the speak() function when she hears something
pet.on("hear", speak);

// Talk to your pet!
hero.say("Hello Kitty");
```

#37a. Buddy's Name A

Level Overview and Solutions

Intro

The peasant wants to know Kitty's name!

However, your pet doesn't have an **event handler** yet!

Use `pet.on("eventName", functionName)` to set a new event for `"hear"` and `sayName`.

Default Code

```
// The peasant wants to know your pet's name.

// Use this function as the handler for "hear" events.
function sayName(event) {
    // The pet will say these in order when this function is called.
    pet.say("My name is Furious Beast.");
    pet.say("But my friends call me Fluffy.");
}
// Use pet.on("eventName", functionName) to add an event listener to your pet.
// In this case use "hear" and sayName with pet.on()!


// You don't need to say anything this time!
// The peasant will do the talking.
```

Overview

An **event handler** is a function that monitors for a specific **event**.

An **event handler** can be created by using the `.on()` method on certain units, like your `pet`!

Use `pet.on("eventType", functionName)` to run the function at `functionName` when the event `"eventType"` occurs. However, this is just an example, as the `"eventType"` event doesn't actually exist! Look at a real example:

Note: Do not include a `()` after the `functionName`! You only want to point to the function, not call it at that exact moment when setting up the **event handler**.

```
// This defines a new function named performTrick!
function performTrick(event) {
    // When this function is called, the pet performs a trick.
    pet.trick();
}
// Using the .on() method, the pet waits for a "hear" event to run performTrick.
pet.on("hear", performTrick);
```

The `"hear"` event occurs when the unit quite literally **hears** something! If someone uses the `.say("string")` method close enough to the unit, the event will **fire** and get **called**. At this point, the code inside the function is run!

In this level the peasant and cows are saying things, so every time they speak, Kitty will say something.

Remember: `pet.on("hear", sayName)` is like saying: **"Pet should `on "hear"` ing something run `sayName`"**.

Buddy's Name A Solution

```
// The peasant wants to know your pet's name.

// Use this function as the handler for "hear" events.
function sayName(event) {
    // The pet will say these in order when this function is called.
    pet.say("My name is Furious Beast.");
    pet.say("But my friends call me Fluffy.");
}
// Use pet.on("eventName", functionName) to add an event listener to your pet.
// In this case use "hear" and sayName with pet.on()!
pet.on("hear", sayName);

// You don't need to say anything this time!
// The peasant will do the talking.
```

#37b. Buddy's Name B

Level Overview and Solutions

Intro

Time to introduce your pet to a new friend!

New friends are always polite to each other, so Kitty should `sayHello` when they `"hear"` something.

Default Code

```
// The pet should greet the hero and peasant.

// Use this function as a handler for "hear" events:
function sayHello(event) {
    // The pet says hello:
    pet.say("Salutations.");
}

// Use the pet's .on("eventType", functionName) method.
// In this level the pet should run sayHello when "hear"ing something.


// Then greet the pet and wait for a response.
hero.say("Hello, my friend!");
```

Overview

An **event handler** is a function that monitors for a specific **event**.

An **event handler** can be created by using the `.on()` method on certain units, like your `pet`!

Use `pet.on("eventType", functionName)` to run the function at `functionName` when the event `"eventType"` occurs. However, this is just an example, as the `"eventType"` event doesn't actually exist! Look at a real example:

Note: Do not include a `()` after the `functionName`! You only want to point to the function, not call it at that exact moment when setting up the **event handler**.

```
// This defines a new function named performTrick!
function performTrick(event) {
    // When this function is called, the pet performs a trick.
    pet.trick();
}
// Using the .on() method, the pet waits for a "hear" event to run performTrick.
pet.on("hear", performTrick);
```

The `"hear"` event occurs when the unit quite literally **hears** something! If someone uses the `.say("string")` method close enough to the unit, the event will **fire** and get **called**. At this point, the code inside the function is run!

In this level the peasant and cows are saying things, so every time they speak, Kitty will say something.

Remember: `pet.on("hear", sayName)` is like saying: **"Pet should `on "hear"` ing something run `sayName`"**.

Buddy's Name B Solution

```
// The pet should greet the hero and peasant.

// Use this function as a handler for "hear" events:
function sayHello(event) {
    // The pet says hello:
    pet.say("Hi-meaw");
}

// Use the pet's .on("eventType", functionName) method.
// In this level the pet should run sayHello when "hear"ing something.
pet.on("hear", sayHello);

// Then greet the pet and wait for a response.
hero.say("Hello, Kitty!");
```

#38. Buddy's Name

Level Overview and Solutions

Intro

You have been given a predefined **event handler** function called `sayName`.

Use the `pet.on(eventType, eventHandler)` method to assign `onHear` as the event handler for `"hear"` events!

```
function onHear(event) {
    pet.say("lol");
}

pet.on("hear", onHear);
```

Default Code

```
// We need to know the name of our new pet.

// Use this function as a handler for "hear" events on pet.
function onHear(event) {
    // Don't change this function.
    pet.say("Meow Woof Meow");
    pet.say("Woof Woof");
    pet.say("Meow");
    pet.say("Meow");
    pet.say("Meow Woof Meow Meow");
}

// Use the pet.on(eventType, eventHandler) method
// Assign the onHear function to handle "hear" events.


// It must be after "pet.on".
hero.say("What's you name, buddy?");
hero.say("Could you repeat it?");
```

Overview

An **event handler** is a function that will be executed when an **event** happens.

You use `pet.on(eventType, eventHandler)` to assign an event handler for an event type, like `"hear"`.

The event handler can be any function you define. It should accept one parameter, the event's data. You'll learn more about the event data later.

For example:

```
function someFunction(event) {
    pet.say("Ahhh");
    pet.say("Bbbbzzz");
}
pet.on("hear", someFunction);
```

Note: You don't use `()` after `someFunction` in `pet.on("hear", someFunction)`. The `()` means that the function is immediately called. Instead we are passing the function as an argument to `.on()` so that it can be called later, when a `"hear"` event happens.

Buddy's Name Solution

```
// We need to know the name of our new pet.

// Use this function as a handler for "hear" events on pet.
function onHear(event) {
    // Don't change this function.
    pet.say("Meow Woof Meow");
    pet.say("Woof Woof");
    pet.say("Meow");
    pet.say("Meow");
    pet.say("Meow Woof Meow Meow");
}

// Use the pet.on(eventType, eventHandler) method
// Assign the onHear function to handle "hear" events.
pet.on("hear", onHear);

// It must be after "pet.on".
hero.say("What's you name, buddy?");
hero.say("Could you repeat it?");
```

#39. Phd Kitty

Level Overview and Solutions

Intro

Let's make a little show for the people in this small village.

Use `pet.say()` to have your pet answer when she hears a question.

Don't forget to use `pet.on()` to assign the `sayTwo` function as the handler for the `"hear"` event.

Default Code

```
// Teach your pet to answer questions!

// Luckily, all the answers are "2"
function sayTwo(event) {
    // Use pet.say() to answer "2"

}

// Use pet.on() to handle "hear" events with sayTwo

// Now relax and watch the show.
hero.say("One plus one is...?");
hero.say("x^3 - 6x^2 + 12x - 8 = 0. What is x...?");
hero.say("How many moons does Mars have...?");
```

Overview

Teach Kitty how to perform special tricks to astound! Preprogram Kitty with the correct answer and impress the onlookers by asking rigged questions.

```
function sayApplesauce(event) {
    pet.say("Applesauce");
}

pet.on("hear", sayApplesauce);

hero.say("What is made up of apples?");
hero.say("What is mashed up into a sauce?");
hero.say("What is improved with a little cinnamon?");
```

Phd Kitty Solution

```
// Teach your pet to answer questions!

// Luckily, all the answers are "2"
function sayTwo(event) {
    // Use pet.say() to answer "2"
    pet.say("2");
}

// Use pet.on() to handle "hear" events with sayTwo
pet.on("hear", sayTwo);
// Now relax and watch the show.
hero.say("One plus one is...?");
hero.say("x^3 - 6x^2 + 12x - 8 = 0. What is x...?");
hero.say("How many moons does Mars have...?");
```

#40. Pet Quiz

Level Overview and Solutions

Intro

Show the university professor that your cat can answer questions.

First, write the function `onHear` from scratch.

Then, add two more questions to impress the professor.

Default Code

```
// Write an event handler function called onHear

// Define the onHear function

    // The pet should say something in onHear.


pet.on("hear", onHear);

hero.say("Do you understand me?");
hero.say("Are you a cougar?");
hero.say("How old are you?");
// Ask two more questions.
```

Overview

Create an event handler function for the pet to execute when they hear something!

Remember how to define a function:

```
function myFunctionName(myParameterName) {
    // do something here
}
```

Pet Quiz Solution

```
// Write an event handler function called onHear

// Define the onHear function
function onHear(e) {
    // The pet should say something in onHear.
    pet.say("Myau");
}

pet.on("hear", onHear);

hero.say("Do you understand me?");
hero.say("Are you a cougar?");
hero.say("How old are you?");
// Ask two more questions.
hero.say("Do you know mathematics?");
hero.say("Do you want to study in the university?");
```

#41. Go Fetch

Level Overview and Solutions

Intro

Your pet knows how to fetch items!

```
var item = hero.findNearestItem();
if(item) {
    pet.fetch(item)
}
```

Default Code

```
// You've been caught in a burl trap!
// Send your pet to fetch the health potions!

function goFetch() {
    // You can use loops in a handler function.
    while(true) {
        var potion = hero.findNearestItem();
        if(potion) {
            // Use pet.fetch to fetch the potion.

        }
    }
}

// When your pet is summoned, it triggers a "spawn" event.
// This tells your pet to run the goFetch function at the start of the level:
pet.on("spawn", goFetch);
```

Overview

Note that your pet can only fetch certain kinds of items, like potions.

Also important on this level is the `"spawn"` event.

Earlier you used the `"hear"` event to have your pet respond to things she heard.

The `"spawn"` event happens only once, at the start of the level, when your pet is summoned.

You can use a `while` loop inside the event handler function, which allows you to have your pet repeat code, even if the `"spawn"` event only happens once.

Go Fetch Solution

```
// You've been caught in a burl trap!
// Send your pet to fetch the health potions!

function goFetch() {
    // You can use loops in a handler function.
    while(true) {
        var potion = hero.findNearestItem();
        if(potion) {
            // Use pet.fetch to fetch the potion.
            pet.fetch(potion);
        }
    }
}

// When your pet is summoned, it triggers a "spawn" event.
// This tells your pet to run the goFetch function at the start of the level:
pet.on("spawn", goFetch);
```

#42. Guard Dog

Level Overview and Solutions

Intro

You can't help the workers across the river in time, but your pet can!

Your pet can `pet.findNearestEnemy()` just like `hero` can!

If your pet sees an enemy, use `pet.say()` to warn the workers!

Default Code

```
// You can't help the peasants across the river.
// But, your pet can!
// Teach your wolf to be a guard dog.

function onSpawn(event) {
    while (true) {
        // Pets can find enemies, too.
        var enemy = pet.findNearestEnemy();
        // If there is an enemy:

            // Then have the pet say something:

    }
}

pet.on("spawn", onSpawn);
```

Overview

Coming soon!

Guard Dog Solution

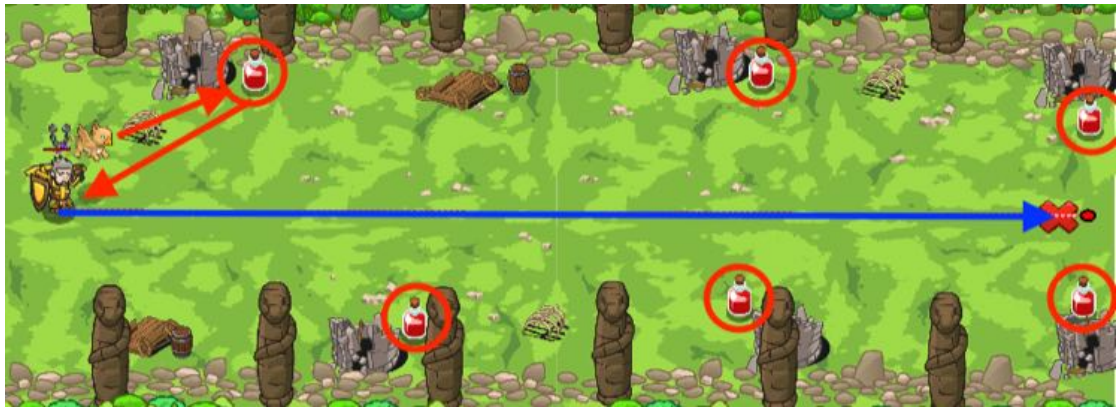
```
// You can't help the peasants across the river.
// But, your pet can!
// Teach your wolf to be a guard dog.

function onSpawn(event) {
    while (true) {
        // Pets can find enemies, too.
        var enemy = pet.findNearestEnemy();
        // If there is an enemy:
        if (enemy) {
            // Then have the pet say something:
            pet.say("Woof!");
        }
    }
}

pet.on("spawn", onSpawn);
```

#43. Long Road

Level Overview and Solutions

Intro

You must escape this ogre trap while poisoned and slowed! Luckily you have your faithful pet to help!



Move to the right, while your pet `fetch`es potions for you. Use `pet.fetch(item)` to carry the potions back to the hero.

Default Code

```
// Move to the right.

// Complete this function:
function onSpawn(event){
    // Inside a while-true loop:

        // Use hero.findNearestItem()

        // If there's an item:

            // Use pet.fetch(item) to fetch the item.

}

// Use pet.on() to assign onSpawn to the "spawn" event

hero.moveXY(78, 35);
```

Overview

You've learnt how to use pet's events in previous levels. If you have problems with this level, then try to return and refresh your skills.

Don't forget to assing an event handler for the certain event:

```
function onSpawn(event) {
    pet.say("First frame!");
}
// It should be outside of the function.
pet.on("spawn", onSpawn);
```

Long Road Solution

```
// Move to the right.

// Complete this function:
function onSpawn(event){
    // Inside a while-true loop:
    while(true) {
        // Use hero.findNearestItem()
        var item = hero.findNearestItem();
        // If there's an item:
        if(item) {
            // Use pet.fetch(item) to fetch the item.
            pet.fetch(item);
        }
    }
}

// Use pet.on() to assign onSpawn to the "spawn" event
pet.on("spawn", onSpawn);
hero.moveXY(78, 35);
```

#44. Forest Jogging

Level Overview and Solutions

Intro

Pets can perform actions independently from the hero.

For example, your pet can move around while your hero is speaking.

Use `pet.moveXY()` in the `onSpawn` function. Then assign `onSpawn` as the **event handler function** for the `"spawn"` **event**.

Default Code

```
// Your pet can use pet.moveXY()

function onSpawn(event) {
    while(true) {
        // First, move to the left X mark:
        pet.moveXY(9, 24);
        // Next, move to the top X mark:
        pet.moveXY(30, 43);
        // Move your pet to the right X mark:

        // Move your pet to the bottom X mark:

    }
}

// Use pet.on() to handle the "spawn" event with onSpawn


// Cheer on your pet!
// Don't remove the commands below.
while(true) {
    hero.say("Good dog!");
    hero.say("You can do it!");
    hero.say("Run Run Run!");
    hero.say("Almost!");
    hero.say("One more lap!");
}
```

Overview

Don't forget to use `while-true` loop inside `onSpawn` function. Otherwise your pet will run only one lap.

```
function onSpawn(event) {
    while(true) {
        pet.moveXY(9, 24);
        ....
    }
}
```

When you assign `onSpawn` for the `"spawn"` event, it's important that `"spawn"` is a string and `onSpawn` is a reference (variable name) for the function.

Forest Jogging Solution

```
// Your pet can use pet.moveXY()

function onSpawn(event) {
    while(true) {
        // First, move to the left X mark:
        pet.moveXY(9, 24);
        // Next, move to the top X mark:
        pet.moveXY(30, 43);
        // Move your pet to the right X mark:
        pet.moveXY(51, 24);
        // Move your pet to the bottom X mark:
        pet.moveXY(30, 5);
    }
}

// Use pet.on() to handle the "spawn" event with onSpawn
pet.on("spawn", onSpawn);

// Cheer on your pet!
// Don't remove the commands below.
while(true) {
    hero.say("Good dog!");
    hero.say("You can do it!");
    hero.say("Run Run Run!");
    hero.say("Almost!");
    hero.say("One more lap!");
}
```

#45. Forest Cannon Dancing

Level Overview and Solutions

Intro

Make your pet to run between the X marks.

First should be the right mark, then the left one.

Write the function `onSpawn` from scratch. Don't forget to use a `while-loop` inside it.

The hero must cheer for the pet!

Default Code

```
// Your pet should run back and forth on the X marks.
// The hero should cheer the whole time!

// Write the event function onSpawn for the pet.
// This function should make the wolf run back and forth:
// First, run to the right mark, then the left one:


pet.on("spawn", onSpawn);
// Cheer up your pet. Don't remove this:
while (true) {
    hero.say("Run!!!");
    hero.say("Faster!");
}
```

Overview

Coming soon!

Forest Cannon Dancing Solution

```
// Your pet should run back and forth on the X marks.
// The hero should cheer the whole time!

// Write the event function onSpawn for the pet.
// This function should make the wolf run back and forth:
// First, run to the right mark, then the left one:
function onSpawn(event) {
    while (true) {
        pet.moveXY(12, 8);
        pet.moveXY(48, 8);
    }
}

pet.on("spawn", onSpawn);
// Cheer up your pet. Don't remove this:
while (true) {
    hero.say("Run!!!");
    hero.say("Faster!");
}
```

#46. Power Peak

Level Overview and Solutions

Intro

Move onto the power discs to gain allies and powers!

Command your pets to move to the unreachable power discs for further power!

Default Code

```
// Welcome to the Course 2 Arena. Defend against waves of ogres!
// Survive longer than your enemy to win!

function petLogic() {
    // Add code to use your pet!
    // Move them to the power discs at the top of the map for powerups.

    pet.say("Command me!");
}

pet.on('spawn', petLogic);
while(true) {
    // Improve your hero's default code!
    // Move to the nearby power discs to spawn units to help or attack!

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

Overview

Power Peak

It's time to *survive*! Command your pet to gain buffs from the mountain power discs while fighting off hordes of munchkins.

Power Peak Solution

```
// Welcome to the Course 2 Arena. Defend against waves of ogres!
// Survive longer than your enemy to win!

function petLogic() {
    // Add code to use your pet!
    // Move them to the power discs at the top of the map for powerups.
    pet.moveXY(66, 74);
}

pet.on('spawn', petLogic);
while(true) {
    // Improve your hero's default code!
    // Move to the nearby power discs to spawn units to help or attack!

    var enemy = hero.findNearestEnemy();
    if(enemy) {
        if(hero.isReady("cleave")) {
            hero.cleave(enemy);
        } else {
            hero.attack(enemy);
        }
    } else {
        hero.moveXY(38, 31)
    }
}
```
