CLOSURES

CLOSURES

BY DEFINITION, A FUNCTION OBJECT RESTS INSIDE A METHOD

SAY WE HAVE A NESTED FUNCTION OBJECT

CLOSURES

OUTER METHOD

VARIABLES LOCAL TO THE OUTER METHOD

NESTED FUNCTION OBJECT
CAN BE ACCESSED FROM HERE..

CLOSURES

EVEN AFTER THE OUTER METHOD CEASES TO EXIST

CLOSURES

OUTER METHOD

VARIABLES LOCAL TO THE OUTER METHOD

NESTED FUNCTION OBJECT

CAN BE ACCESSED FROM HERE..
EVEN AFTER THE OUTER METHOD
CEASES TO EXIST

CLOSURES

OUTER METHOD

VARIABLES LOCAL TO THE OUTER METHOD

NESTED FUNCTION OBJECT
CAN BE ACCESSED FROM HERE..

EVEN AFTER THE OUTER METHOD
CEASES TO EXIST

CLOSURE = NESTED FUNCTION OBJECT + VARIABLES LOCAL TO THE OUTER METHOD

CLOSURE = NESTED FUNCTION

+

VARIABLES LOCAL TO THE OUTER SCOPE

SCOPE IS THE MORE TECHNICAL, AND GENERAL TERM FOR THE OUTER METHOD

CLOSURE = NESTED FUNCTION

+

VARIABLES LOCAL TO THE OUTER SCOPE

SCOPE IS THE MORE TECHNICAL, AND GENERAL TERM FOR THE OUTER FUNCTION

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

REMEMBER THIS EQUATION, AND WE WILL BE JUST FINE!

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

THIS SEEMS LIKE MAGIC - AND IT IS.

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #1: THE NESTED FUNCTION CAN ACCESS THE REFERENCING ENVIRONMENT - EVEN THOUGH THOSE VARIABLES ARE OUTSIDE THE SCOPE

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #1: THE NESTED FUNCTION CAN ACCESS THE REFERENCING ENVIRONMENT - EVEN THOUGH THOSE VARIABLES ARE OUTSIDE THE SCOPE

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #1: THE NESTED FUNCTION CAN ACCESS THE REFERENCING ENVIRONMENT - EVEN THOUGH THOSE VARIABLES ARE OUTSIDE THE SCOPE

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES AROUND THAT REFERENCING ENVIRONMENT EVEN AFTER THE SCOPE HAS "GONE AWAY"!

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES AROUND THAT REFERENCING ENVIRONMENT EVEN AFTER THE SCOPE HAS "GONE AWAY"!

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES AROUND THAT REFERENCING ENVIRONMENT EVEN AFTER THE SCOPE HAS "GONE AWAY"!

CLOSURE = NESTED FUNCTION +

VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

MAGIC TRICK #2: THE NESTED FUNCTION CARRIES AROUND THAT REFERENCING ENVIRONMENT EVEN AFTER THE SCOPE HAS "GONE AWAY"!

LETS' APPLY OUR EQUATION TO AN EXAMPLE

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

LET'S GO BACK TO THE GREETING EXAMPLE

```
def main (args: Array[String]){

  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}


def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```

WE HAVE A METHOD THAT RETURNS A FUNCTION

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

```
  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")
}
```

A METHOD THAT RETURNS A FUNCTION

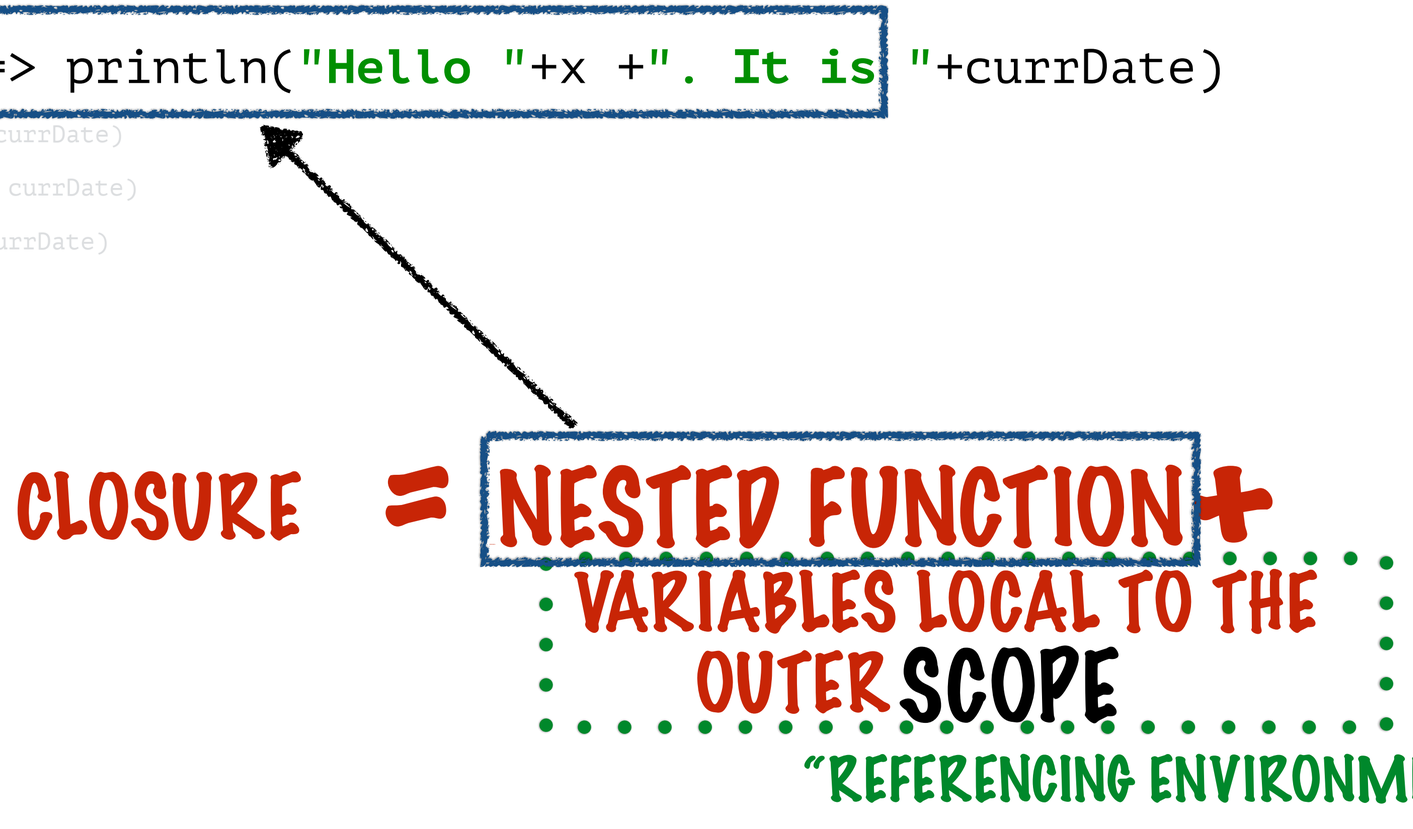
```
def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

```
  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}
```

```
def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```



```
  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}
```

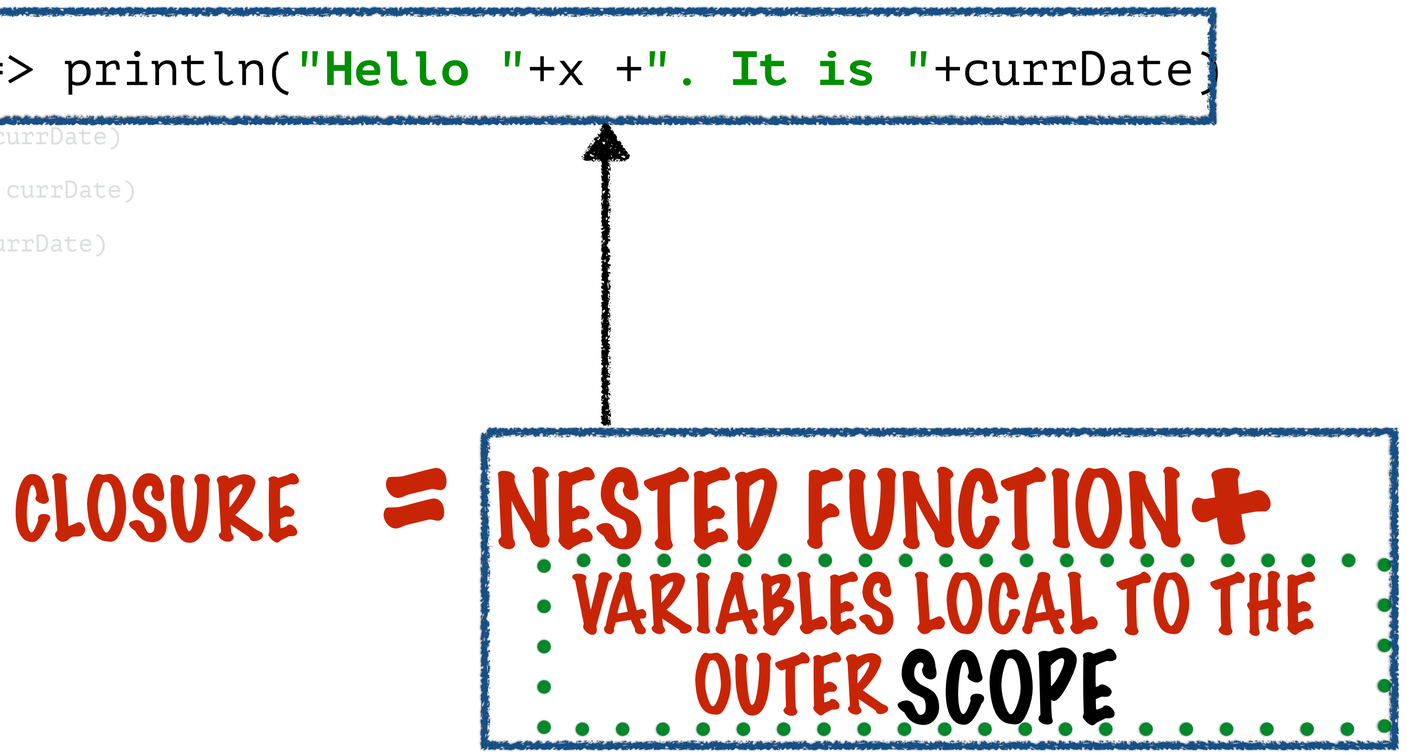
```
def greeting(lang: String)= {
  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE

"REFERENCING ENVIRONMENT"

```
  val greetEnglish = greeting("English")
  greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}
```

```
def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```



```
def main (args: Array[String]){

    val greetEnglish = greeting("English")
    greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}


def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```

WHEN THE GREETING METHOD IS CALLED IT RETURNS A FUNCTION OBJECT

WHICH IS A CLOSURE

```
def main (args: Array[String]){

    val greetEnglish = greeting("English")
    greetEnglish("Swetha")

  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}


def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```

METHOD IS CALLED HERE, WITHIN THE METHOD THE CURRDATE VARIABLE IS CREATED

A CLOSURE IS RETURNED AND STORED IN GREETENGLISH

```
def main (args: Array[String]){

  val greetEnglish = greeting("English")
  greetEnglish("Swetha")
  val greetSpanish = greeting("Spanish")
  greetSpanish("Janani")

}


def greeting(lang: String)= {

  val currDate = Calendar.getInstance().getTime().toString

  lang match {

    case "English" => (x: String) => println("Hello "+x +". It is "+currDate)

    case "Hindi" => (x: String) => println("Namaste "+x +". It is "+ currDate)

    case "French" => (x: String) => println("Bonjour "+x +". It is "+ currDate)

    case "Spanish" => (x: String) => println("Hola "+x +". It is "+ currDate)

  }
}
```

HERE THE GREETING METHOD HAS CEASED TO EXIST

BUT, THE CLOSURE CAN STILL USE THE CURRDATE VARIABLE

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE "REFERENCING

WHY ARE CLOSURES SO IMPORTANT?

ITS BECAUSE CLOSURES CARRY THE ENVIRONMENT AROUND THEM WHEREVER THEY GO

CLOSURE = NESTED FUNCTION + VARIABLES LOCAL TO THE OUTER SCOPE "REFERENCING

WHY ARE CLOSURES SO IMPORTANT?

IN A DISTRIBUTED ENVIRONMENT, CLOSURES MAKE IT EASIER TO SHIP CODE TO DIFFERENT NODES IN A CLUSTER

CURRYING

LET'S SAY WE ARE PLAYING A GAME CALLED "IN SIGHT"

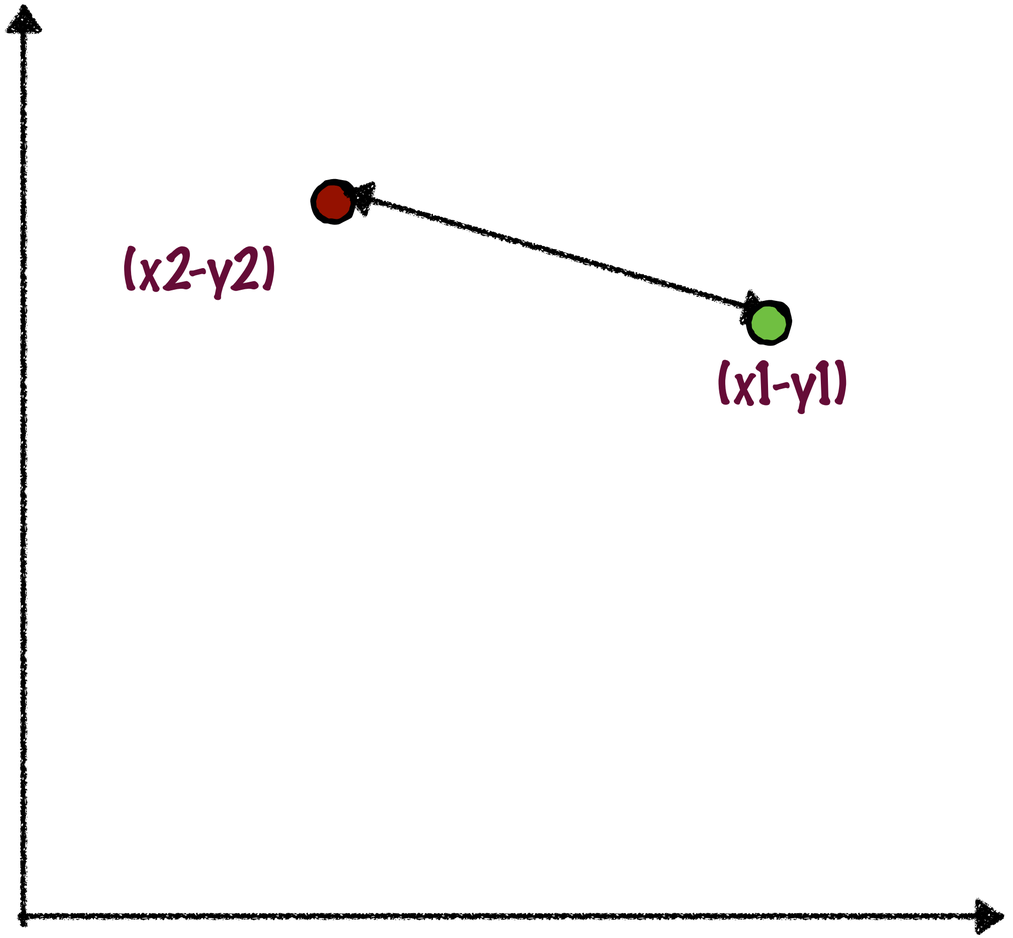
CURRYING

"IN SIGHT"

YOU ARE GIVEN

1. A PERSON'S CO-ORDINATES
2. AN ITEM'S CO-ORDINATES

DETERMINE WHETHER THE ITEM IS IN VISIBLE RANGE OF THE PERSON

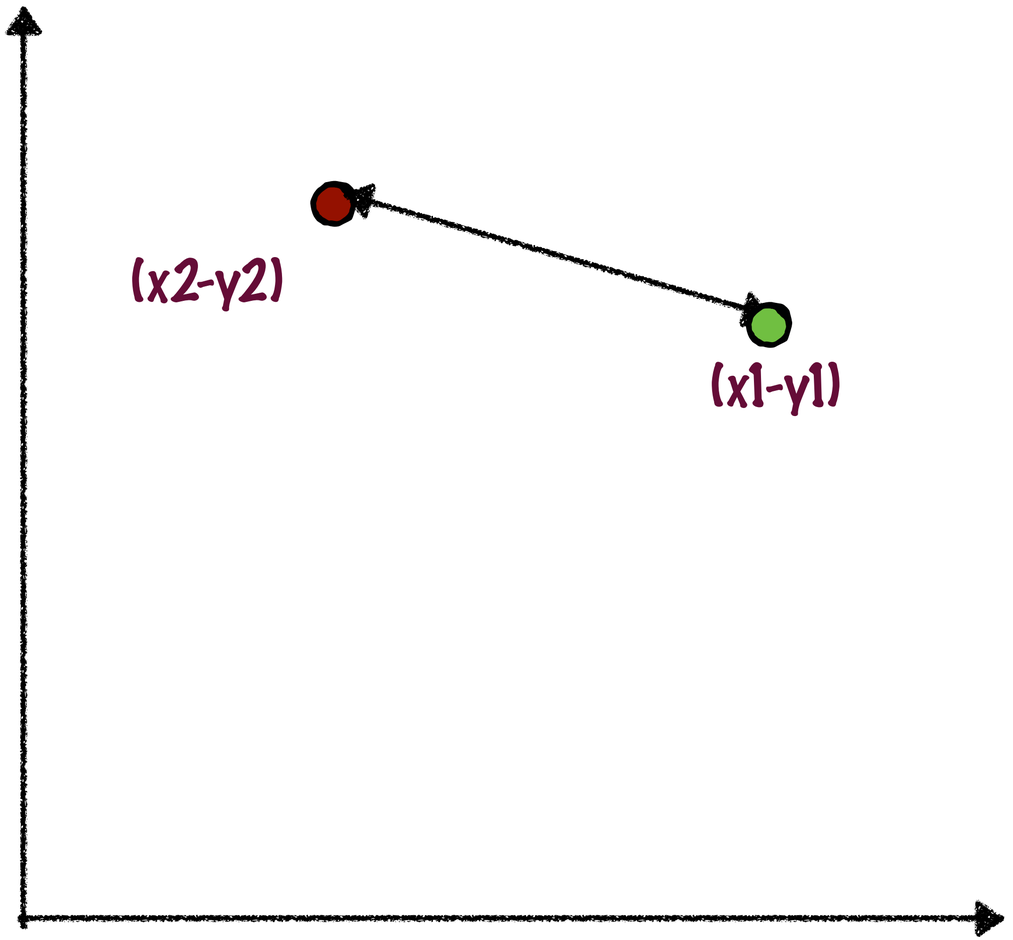


CURRYING

"IN SIGHT"

DETERMINE WHETHER THE ITEM IS IN VISIBLE RANGE OF THE PERSON

THE VISIBLE RANGE IS A CONSTANT DEFINED FOR A PERSON

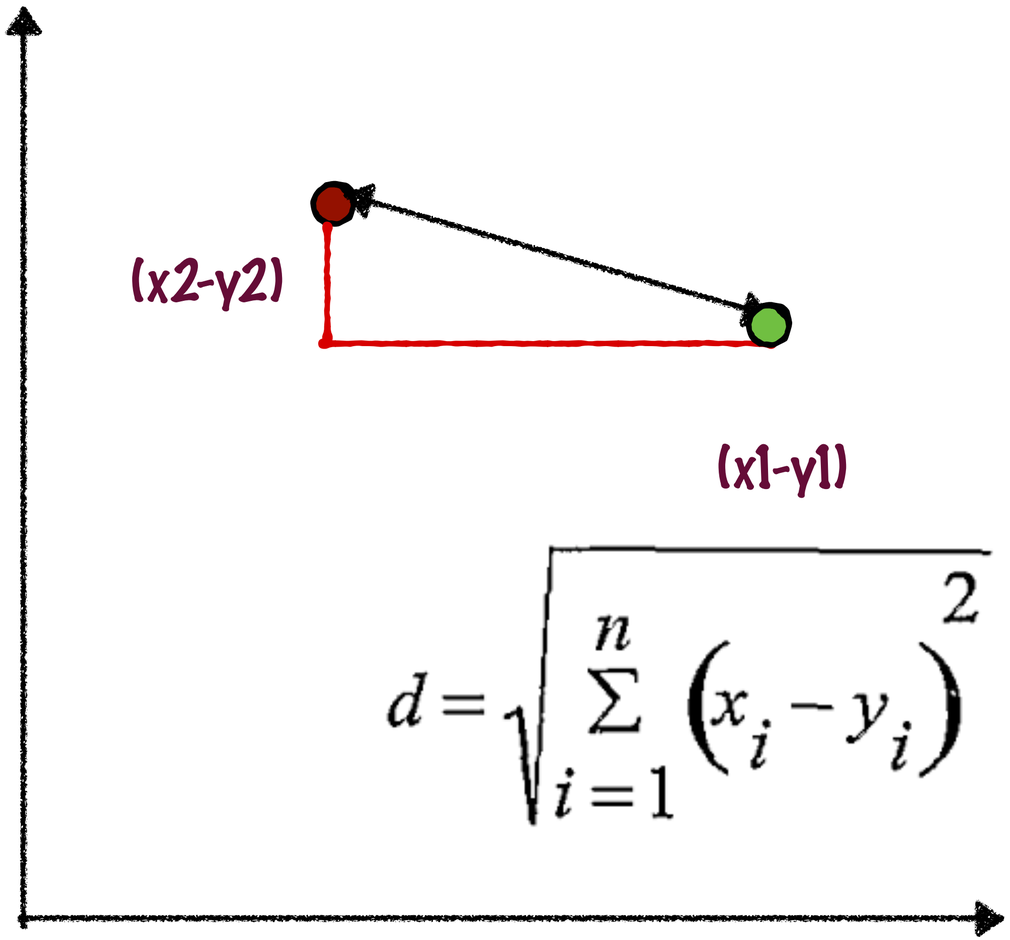


CURRYING

"IN SIGHT"

DETERMINE WHETHER THE ITEM IS IN VISIBLE RANGE OF THE PERSON

YOU CAN COMPUTE THE DISTANCE BETWEEN THE 2 POINTS

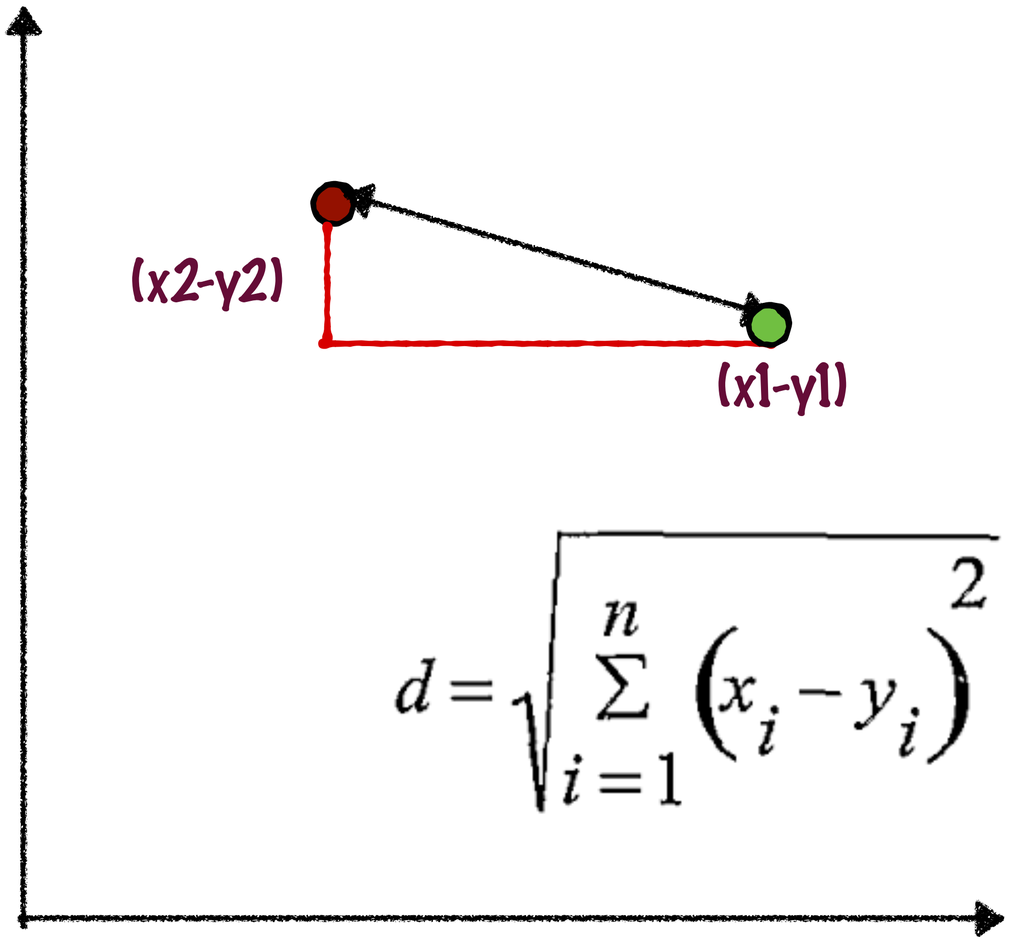


CURRYING

"IN SIGHT"

DETERMINE WHETHER THE ITEM IS IN VISIBLE RANGE OF THE PERSON

COMPARE THE DISTANCE TO THE VISIBLE RANGE TO SEE IF THE ITEM IS "IN SIGHT"



FIRST LET'S SET UP
CLASSES TO
REPRESENT
PERSONS AND ITEMS

CURRYING

"IN SIGHT"

```
class Person(xc: Double,yc: Double,r: Double) {

  val x = xc
  val y = yc
  val range = r

}
```

A PERSON IS DESCRIBED BY THEIR (X,Y) CO-ORDINATES AND THEIR VISIBLE RANGE

CURRYING

"IN SIGHT"

```
class Person(xc: Double,yc: Double,r: Double) {

  val x = xc
  val y = yc
  val range = r

}
```

SCALA HAS SHORTHAND TO CREATE SUCH A CLASS

```
case class Person(
x: Double,
y: Double,
range: Double)
```

"IN SIGHT"

```
case class Person(
x: Double,
y: Double,
range: Double)
```

CASE CLASSES ARE SPECIAL TYPE OF CLASSES THAT BE SETUP USING JUST A CONSTRUCTOR

SCALA WILL INFER THAT THE CLASS ENCAPSULATES VARIABLES WITH THE SAME NAMES AS THOSE DEFINED THE CONSTRUCTOR

"IN SIGHT"

```
case class Person(
x: Double,
y: Double,
range: Double)
```

CASE CLASSES AUTOMATICALLY COME WITH `toString`, `hashCode` AND SOME OTHER METHODS IMPLEMENTED

ALONG WITH BEING SYNTACTIC SUGAR, CASE CLASSES ALSO HELP WITH PATTERN MATCHING, BUT WE WON'T GO INTO THAT HERE

"IN SIGHT"

BACK TO OUR INSIGHT FUNCTION

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <= pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

BACK TO OUR INSIGHT FUNCTION

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <= pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

SET UP BOTH THE PERSON AND ITEM CLASSES

CURRYING

"IN SIGHT"

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <= pow(person.range,2)
  }
```

```
val p1 = new Person(0,1,2)
```

```
val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

SET UP A FEW PERSON AND ITEM OBJECTS

CURRYING

"IN SIGHT"

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <=
pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
```

A FUNCTION THAT RETURNS TRUE IF THE ITEM IS IN THE PERSON'S VISIBLE RANGE

CURRYING

"IN SIGHT"

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <=
pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
```

THE FUNCTION TAKES A PERSON AND AN ITEM

CURRYING

"IN SIGHT"

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <=
pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
```

THE SQUARE OF THE DISTANCE BETWEEN THE 2 POINTS

CURRYING

"IN SIGHT"

```
case class Person(x: Double,y: Double,range: Double);
case class Item(x: Double,y: Double);

val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <=
pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
```

THE SQUARE OF THE PERSON'S VISIBLE RANGE

CURRYING

"IN SIGHT"

```
val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

```
inSight(p1,i1)
inSight(p1,i2)
inSight(p1,i3)
inSight(p1,i4)
```

WE CALL THE FUNCTION TO CHECK WHETHER EACH ITEM IS WITHIN VISIBLE RANGE

CURRYING

```
val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <= pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

```
inSight(p1,i1)
inSight(p1,i2)
inSight(p1,i3)
inSight(p1,i4)
```

"IN SIGHT"

THE SAME ARGUMENT IS PASSED IN WITH EACH FUNCTION CALL

CURRYING

```
val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <= pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

```
inSight(p1,i1)
inSight(p1,i2)
inSight(p1,i3)
inSight(p1,i4)
```

"IN SIGHT"

IMAGINE THIS FUNCTION DID SOME COMPLEX CALCULATIONS WITH THE MEMBERS OF P1

THAT CALCULATION WOULD HAVE TO BE REPEATED EVERY TIME

CURRYING

```
val inSight
= (person: Person, item: Item ) => {
    pow( person.x-item.x, 2 ) + pow( person.y-item.y, 2 ) <= pow(person.range,2)
  }

val p1 = new Person(0,1,2)


val i1 = new Item(1,1)
val i2 = new Item(3,1)
val i3 = new Item(1,2)
val i4 = new Item(2,0)
```

```
inSight(p1,i1)
inSight(p1,i2)
inSight(p1,i3)
inSight(p1,i4)
```

"IN SIGHT"

WHAT IF WE COULD HAVE A FUNCTION SPECIFIC TO THE PERSON?

THIS IS EXACTLY THE IDEA BEHIND CURRYING

CURRYING

"IN SIGHT"

```
val inSightCurried = (person: Person) => (item: Item ) => inSight(person,item)
```

INSIGHTCURRIED IS A FUNCTION

CURRYING

"IN SIGHT"

```
val inSightCurried = (person: Person) => (item: Item ) => inSight(person,item)
```

IT TAKES A PERSON AS AN ARGUMENT

CURRYING

"IN SIGHT"

```
val inSightCurried = (person: Person) => (item: Item ) => inSight(person,item)
```

RETURNS ANOTHER FUNCTION

CURRYING

"IN SIGHT"

```
val inSightCurried = (person: Person) => (item: Item ) => inSight(person,item)
```

```
val inSightForP1= inSightCurried(p1)
```

A NEW FUNCTION OBJECT WHICH ONLY NEEDS 1 ARGUMENT

CURRYING

"IN SIGHT"

```
val inSightCurried = (person: Person) => (item: Item ) => inSight(person,item)
```

```
val inSightForP1= inSightCurried(p1)
```

```
inSightForP1(i1)
inSightForP1(i2)
inSightForP1(i3)
inSightForP1(i4)
```

USE THIS TO FIND WHETHER EACH OF THE ITEMS ARE IN VISIBLE RANGE OF P1

CURRYING

LET'S SUMMARIZE

WHENEVER YOU HAVE A FUNCTION
WITH MULTIPLE ARGUMENTS

```
(x,y,z) => F(x,y,z)
```

YOU CAN CREATE A CHAIN OF FUNCTIONS,
EACH WITH ONLY 1 ARGUMENT

```
x => y => z => F(x,y,z)
```

CURRYING

```
x => y => z => F(x,y,z)
```

FIRST FUNCTION TAKES ONLY X AS AN ARGUMENT

CURRYING

```
x => y => z => F(x,y,z)
```

RETURNS A FUNCTION WHICH TAKES Y AS AN ARGUMENT

THIS FUNCTION TAKES Y

CURRYING

```
x => y => z => F(x,y,z)
```

RETURNS A FUNCTION WHICH TAKES Z AS AN ARGUMENT

THIS FUNCTION TAKES Z

CURRYING

```
x => y => z => F(x,y,z)
```

RETURNS A VALUE

THIS PROCESS OF CONVERTING A FUNCTION INTO A CHAIN OF FUNCTIONS IS CALLED CURRYING

CURRYING

```
x => y => z => F(x,y,z)
```

CURRYING IS A COMMONLY USED FUNCTIONAL PROGRAMMING CONSTRUCT

CURRIED METHODS

CURRIED METHODS

CURRIED METHODS CAN TAKE MULTIPLE ARGUMENTS

IF ONLY 1 ARGUMENT IS GIVEN, THEY RETURN A FUNCTION OBJECT

CURRIED METHODS

```
def curriedSum(x: Int)(y: Int) = x + y
```

A CURRIED METHOD TO COMPUTE THE SUM OF 2 INTEGERS

```
scala> curriedSum(1)(2)
res1: Int = 3
```

Curried Methods

```
def curriedSum(x: Int)(y: Int) = x + y
```

PARTIALLY APPLYING THIS METHOD RETURNS A FUNCTION OBJECT

```
scala> val twoPlus = curriedSum(2)_

twoPlus: Int => Int = <function1>
```