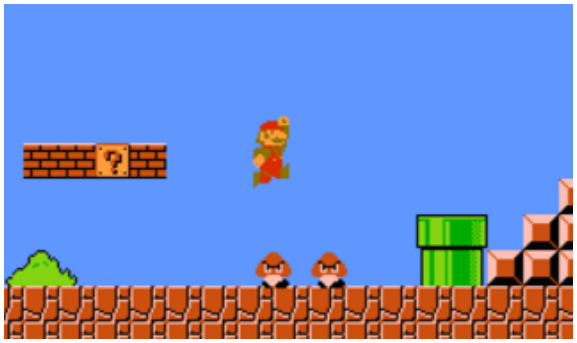
Binary, Data Types, Wrapper Classes

For this lecture, we will cover:

- binary
- expand upon our primative data types
- discuss the wrapper classes related to the primative data types

8 Bit	16 bit
	

> Discussion
> What is a bit? and a byte? - they are not cookies!

Binary Bits

Binary – two state system

- 1 for on
- 0 for off

Bit

- Each 0 and 1 is called a bit
- 8 bits is called a byte
- Contains 255 states (128+64+32+16+8+4+2+1)

Examples:

- ~1000 bytes is a kilobyte
- ~1,000,000 bytes is a megabyte
- A song is often 3-5 megabytes

Binary Practice

- Every bit is a exponential of 2
  - 1 + 2 + 4 + 8, etc

Take

$$1001$$

$$= 2^3 * 1 + 2^2 * 0 + 2^1 * 0 + 2^0 * 1$$

$$= 8 + 0 + 0 + 1$$

$$= 9$$

> Math reminder
> Any number to the power of 0 is 1.

This means with 4 bits, I can represent 16 different states! (0-15). With 5 bits, I get to double that, representing 32 states (0-31).

This double of every bit is huge!

> Fun Tip:
> Knowing binary means you can count to 32 on one hand, and 1023 on two hands

In [2]:
```
public static int binaryToDecimalConverter(String binaryStr) {
    int dec = Integer.parseInt(binaryStr, 2); // convert the string using base 2
    return dec;
}
```

In [3]:
```
int answer = binaryToDecimalConverter("011");
System.out.println(answer)
```

```
3
```

In [4]:
```
int answer = binaryToDecimalConverter("111");
System.out.println(answer)
```

```
7
```

In [5]:
```
int answer = binaryToDecimalConverter("1011");
System.out.println(answer)
```

```
11
```

Now, let's flip it!

In [6]:
```
System.out.print(Integer.toBinaryString(8));
```

```
1000
```

In [7]:
```
System.out.print(Integer.toBinaryString(31));
```

```
11111
```

More Data Types

When we declare a `type`

- we are requesting a certain number of bits!

The primative types we know are:

- boolean
  - 1 bit - yes or no
- char
  - 16 bit - converted to characters via ascii/unicode table
- int
  - 32 bit
- double
  - 64 bit with floating point (decimal)

However, there is a range of primative types, based on your need.

- java centers the 'states' on zero,
- so for example a byte is -128-127, isntead of 0-255

Type	Size	Supported number range	Category
byte	8 bits	-128 to 127	Whole Number
short	16 bits	-32,768 to 32,767	Whole Number
int	32 bits	-2,147,483,648 to 2,147,483,647	Whole Number
float	32 bits	-3.4 x 10^{38} to 3.4 x 10^{38}	Floating Point
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Whole Number
double	64 bits	-1.7 x 10^{308} to 1.7 x 10^{308}	Floating Point

Note: we don't expect you to memorize the number range, but being able to recognize the different sizing is good.

> Casting:
> This is also why an int will automatically convert to a double, but a double can't convert to an int without explict casting - possible loss of information!

In [8]:
```
short red = 127;   // to think about, why did I have to use short, instead of byte?
short green = 135;
short blue = 255;

String htmlCode = String.format("#%02X%02X%02X", red, green, blue);

System.out.println(htmlCode);
```

```
#7F87FF
```

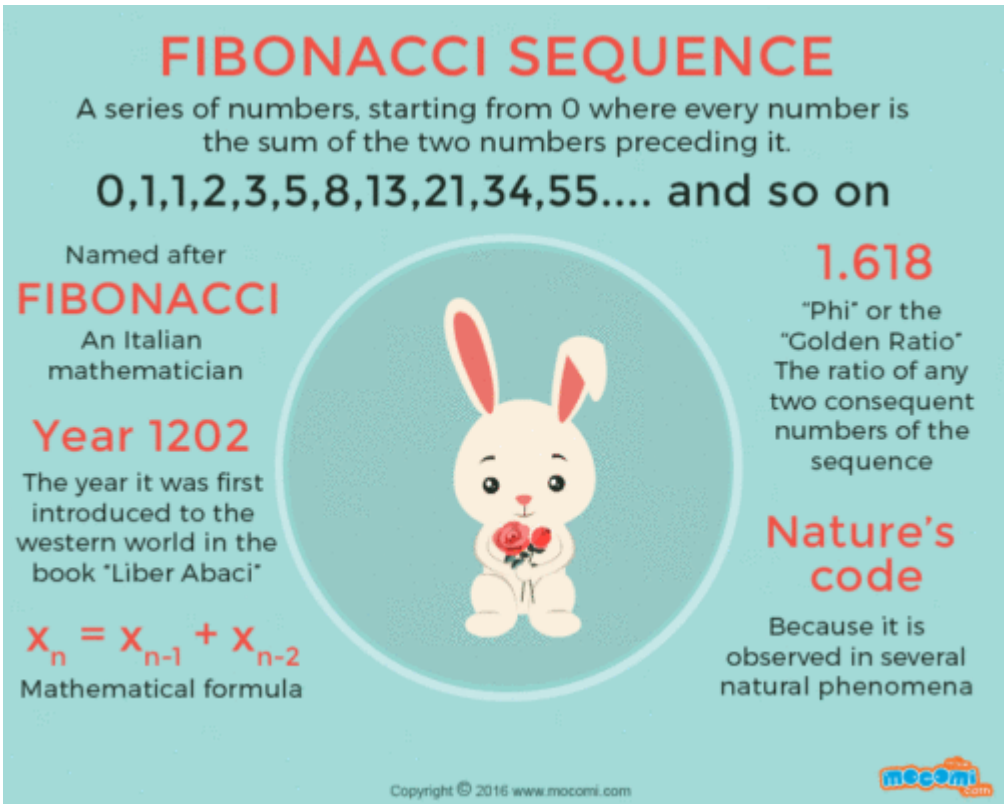
In [9]:
```
public static int fib(int n) {
    int rtn = 0;
    for(int i = 0, current = 1; i < n-1; i++) {
```

```
        System.out.print(rtn + " ");
        int tmp = rtn;
        rtn += current;
        current = tmp;
    }
    System.out.println(rtn);
    return rtn;

}

int fibAns = fib(6);
```

```
0 1 1 2 3 5
```



Let's try it with a variety of values!

In [10]:
```
fibAns = fib(24)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2865
7
```

In [11]:
```
fibAns = fib(47)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2865
7 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227
465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 70140
8733 1134903170 1836311903
```

Uh oh, we have a problem!

$fib(48) = 1,134,903,170 + 1,836,311,903$

$fib(48) = 2,971,215,073$

However, the max size we can hold in an `int` is 2,147,483,647!

Let's see what happens!

In [12]:
```
fibAns = fib(48)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2865
7 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227
465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 70140
8733 1134903170 1836311903 -1323752223
```

$-1,323,752,223$ - WHAT!

The value 'overflows' and wraps around between the max to the min of int! Not at all what we would expect, and causes unusual situations.

The fix? Let's go back to our types!

- short is less bits than int
- long is more bits

seems like we should have used `long`

In [13]:
```
public static long bigFib(int n) {
    long rtn = 0;
    long current = 1;
    for(int i = 0; i < n-1; i++) {
        System.out.print(rtn + " ");
        long tmp = rtn;
        rtn += current;
        current = tmp;
    }
    System.out.println(rtn);
    return rtn;

}

long big = bigFib(48);
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2865
7 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227
465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 70140
8733 1134903170 1836311903 2971215073
```

In [14]:
```
big = bigFib(80);
System.out.println();
System.out.println();
// but what about?
long big2 = bigFib(100);
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2865
7 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227
465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 70140
8733 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025 20365011074 3
2951280099 53316291173 86267571272 139583862445 225851433717 365435296162 59128672987
9 956722026041 1548008755920 2504730781961 4052739537881 6557470319842 10610209857723
17167680177565 27777890035288 44945570212853 72723460248141 117669030460994 190392490
709135 308061521170129 498454011879264 806515533049393 1304969544928657 2111485077978
050 3416454622906707 5527939700884757 8944394323791464 14472334024676221
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 2865
7 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227
465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 70140
8733 1134903170 1836311903 2971215073 4807526976 7778742049 12586269025 20365011074 3
2951280099 53316291173 86267571272 139583862445 225851433717 365435296162 59128672987
9 956722026041 1548008755920 2504730781961 4052739537881 6557470319842 10610209857723
17167680177565 27777890035288 44945570212853 72723460248141 117669030460994 190392490
709135 308061521170129 498454011879264 806515533049393 1304969544928657 2111485077978
050 3416454622906707 5527939700884757 8944394323791464 14472334024676221 234167283484
67685 37889062373143906 61305790721611591 99194853094755497 160500643816367088 259695
496911122585 420196140727489673 679891637638612258 1100087778366101931 17799794160047
14189 2880067194370816120 4660046610375530309 7540113804746346429 -6246583658587674878
 1293530146158671551 -4953053512429003327 -3659523366270331776 -8612576878699335103
6174643828739884737 -2437933049959450366
```

Data Types - Summary

They are useful to know!

- For the most part:
  - we use `int` and `double`
- However for known cases
  - Alright to change it
  - long is especially common
    - but we have all dealt with a program sucking the computers memory!

What happens on **really** big numbers?

- BigInteger class - handles an "unlimited" size, but at an overhead cost
- BigDecimal class - same, but with floating points
  - BigDecimal is more accurate than `double`, so recommended for currency calculations

Wrapper Classes

The problem: primatives lack actions / functionality The solution: wrapper classes!

For every primative, there is a matching Object version.

primative	Object
boolean	Boolean

primative	Object
byte	Byte
short	Short
char	Character
int	Integer
float	Float
long	Long
double	Double

Like primatives:

```
    * Integer, Character, Double  - are the most used
```

Wrapper Static Functionality:

- There a many static methods in each class
  - Integer.parseInt(String) - takes a String, converts it to an int
  - Double.parseDouble(String) - takes a String, converts it to a double
  - Character.isWhitespace(char) - returns true if it is whitespace (space, tab, etc)
  - Character.isLetterOrDigit(char) - returns true if it is a letter or digit
  - Character.isDigit(char) - return true if it is a digit

Let's use them!

Task 1 : Loop and Character.isDigit(char) practice

Write a loop that takes a String, and only returns the digits in that String

example:

```
811-111-111
```

would become

```
811111111
```

String.length() and String.charAt(x) will be helpful!

In [15]:
```
String test = "811-111-111";
String numOnly = "";
for(int i =0; i < test.length(); i++) {
    if(Character.isDigit(test.charAt(i))) {
        numOnly += test.charAt(i);
    }
}

System.out.println(numOnly);
```

```
811111111
```

Let's also use try some more

Task 2 - Convert to double

Write a program that takes in two numbers

- the first number is the "left" half of a decimal
- the second number is the "right" half of a decimal
- combine the two halves, and return the double value!

See method convert halves to double in the in class activity.

In [17]:
```
String one = "111";
String two = "456";

double dbl = Double.parseDouble(String.format("%s.%s", one, two));

System.out.println(dbl / 2);
```

```
55.728
```

Autoconversation / Boxing and unboxing

Since going back and forth between Integer and int (and other wrapper classes to their type) is so common. Java added boxing/unboxing.

Boxing:

- Wrapping a primative in its Object wrapper class Unboxing
- Pulling the primative out of the object.

For example:

In [21]:
```
Integer x = 10;
System.out.println(x.toString().charAt(0)); // only valid because x is the Object "Int

// butw e can also do the following

int value = x + 32; // converted x to an int, added 32, stored in value
System.out.println(value);
```

```
1
42
```

Why not always use the objects?

- valid in some languages (python, kotlin)
- cost overhead with java and similiar languages
  - which slows down your program.

We will come across the most important use of boxing/unboxing on Monday! Have a great weekend.