Chapter 1.1: Computers

Object Oriented Programming with Java

Basic parts | Binary numbers | Two's complement | Floating point | ASCII and Unicode | Strings | Stored program | von Neumann architecture

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/java/slides/Chapter_01_01_Computers.html

Learning Outcomes

Hardware

Explain what the CPU, memory, disk and bus do; trace how a program runs through the fetch–decode–execute cycle.

Numbers

Convert between binary, octal, hexadecimal and decimal; add in binary and form negative numbers with two's complement.

Text

Explain how ASCII, Unicode and UTF-8 differ; show how a string is stored in memory as numbers.

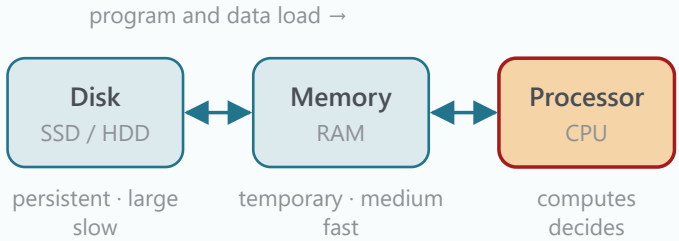
Connecting to Java

Explain the limits of `byte` and `int` and overflow via two's complement; work with bases in Java; map assembly instructions to machine code.

COMPUTER BASICS

The Basic Parts of a Computer

- **Processor** (CPU, *central processing unit*): the “brain” of the computer; it does all **arithmetic** and **logical** work.
- **Memory** (RAM, *random access memory*): holds the running program and its data **temporarily**. Its size and access speed directly affect how fast the computer is.
- **Disk** (HDD / SSD): stores data **persistently**; nothing is lost when the power goes off.



Kitchen analogy: the cook is the CPU, the counter is RAM (small but within reach), the pantry is the disk (large but far away). The cook brings ingredients from the pantry to the counter and works there.

COMPUTER BASICS

How a Computer Works, and Programming

The basic steps:

1. The processor receives **instructions**.
2. It carries them out **in order**.
3. The results go to the user or to other software.

Programming is writing instructions, in a form the computer understands, so that it performs the task we want.

Our first Java program: a single instruction, "print to the screen".

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Output:

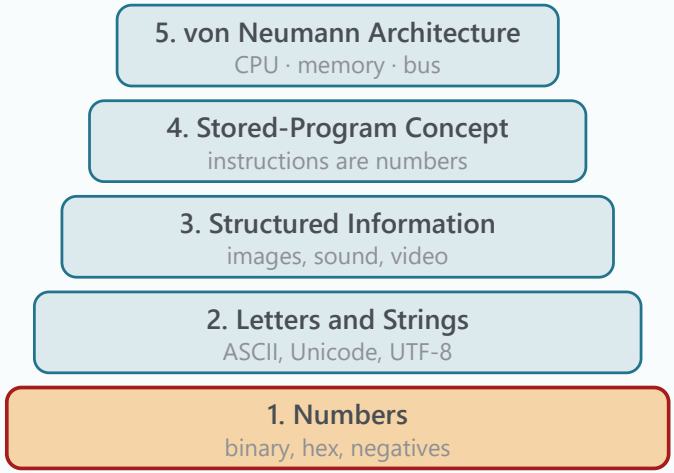
```
Hello, World!
```

Instructions written in a language like Java run only after they are translated into machine code the processor understands.

How Do You Build a Computer?

We build the computer layer by layer, each resting on the one below:

- First we turn electrical signals into **numbers**.
- Numbers represent **letters**; letters make up **text**.
- Sequences of numbers store structured information such as **images, sound and video**.
- **Instructions too** are stored in memory as numbers.
- The **von Neumann architecture** ties it all together.

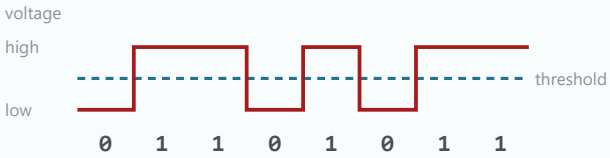


From Electrical Signal to Bit

- At bottom, a computer processes **electrical signals** by fixed rules; those rules give rise to everything it does.
- Once signals are tied to **numbers and symbols** people understand, the computer can give us meaningful information.
- Signals are handled in **binary** (base 2): each digit, or **bit** (*binary digit*), is either 0 or 1.
- Every operation is built on this binary foundation.

0	1
false	true
Off	On
Low voltage	High voltage

Why two states? Telling "voltage / no voltage" apart reliably despite noise is far easier than distinguishing ten voltage levels.



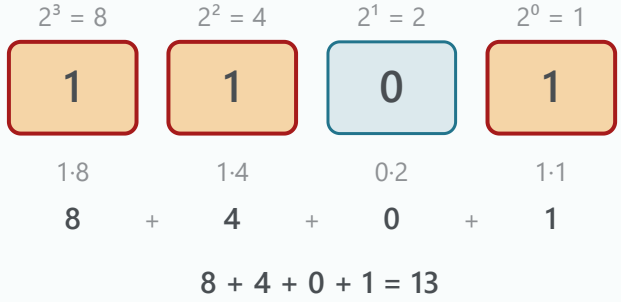
8 bits = 1 **byte**. Memory and file sizes are measured in bytes: 1 KB ≈ 1000 bytes, 1 KiB = 1024 bytes.

Place Value: $13 = 1101_2$

- An integer is written as a sum of **powers of 2**, each used 0 or 1 times.
- Four consecutive powers (2^0, 2^1, 2^2, 2^3) cover the integers **0–15**.
- $13_{10} = 1{\cdot}2^3 + 1{\cdot}2^2 + 0{\cdot}2^1 + 1{\cdot}2^0 = \mathbf{1101_2}$

The same rule in other bases:

Decimal (10)	$603_{10} = 6{\cdot}10^2 + 0{\cdot}10^1 + 3{\cdot}10^0$
Octal (8)	$207_8 = 2{\cdot}8^2 + 0{\cdot}8^1 + 7{\cdot}8^0 = 135_{10}$
Hexadecimal (16)	$FF_{16} = 15{\cdot}16^1 + 15{\cdot}16^0 = 255_{10}$



The subscript gives the base: 1101_2 is read “one one zero one, binary”, not “one thousand one hundred one”.

Any Base, and the Limits of Bits

- With base **b ≥ 2** and **d** digits you can write every integer from 0 to $\mathbf{b^d - 1}$.
- Digits run from 0 to **b − 1** (0–1 in binary, 0–7 in octal, 0–9 and A–F in hex).
- A 64-bit computer can count, unsigned, up to $2^{64} - 1 =$ **18,446,744,073,709,551,615**.

Bit	Largest ($2^d - 1$)
4	15
8	255
16	65,535
32	4,294,967,295

Java's integer types are **signed**: half of the bit patterns go to negatives.

```
System.out.println(Byte.MAX_VALUE);     // 8 bit
System.out.println(Short.MAX_VALUE);    // 16 bit
System.out.println(Integer.MAX_VALUE); // 32 bit
System.out.println(Long.MAX_VALUE);     // 64 bit
// all 64 bits 1, read unsigned:
System.out.println(Long.toUnsignedString(-1L));
```

Output:

```
127
32767
2147483647
9223372036854775807
18446744073709551615
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
   int n = 13;
3  String bits = "";
4  while (n > 0) {
5    int rem = n % 2;
6    System.out.println(n + " % 2 = " + rem);
7    bits = rem + bits;
8    n = n / 2;
9  }
10 System.out.println("Result: " + bits);
11 }
```

n ÷ 2 rem

bits ""

Start with `n` = 13.

Step 1 / 16

VARIABLES

`n = 13`

`bits = –`

`rem = –`

OUTPUT

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
▶    String bits = "";
4    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```

n ÷ 2 rem

bits ""

The result string starts empty: `bits = ""`.

Step 2 / 16

VARIABLES

`n = 13`

`bits = ""`

`rem = –`

OUTPUT

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
▶    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```

n ÷ 2 rem

bits ""

`13 > 0` is true → enter the loop.

Step 3 / 16

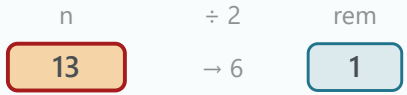
VARIABLES

- `n = 13`
- `bits = ""`
- `rem = –`

OUTPUT

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
▶      int rem = n % 2;
▶      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



bits `""`

Remainder `13 % 2 = 1` : the **rightmost** bit not yet written.

Step 4 / 16

VARIABLES

`n = 13`

`bits = ""`

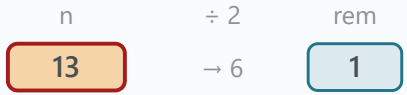
`rem = 1`

OUTPUT

```
13 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
       bits = rem + bits;
       n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



bits **1**

The remainder goes in **front** of the string → "1"; n is halved: 13 / 2 = 6 (integer division).

Step 5 / 16

VARIABLES

`n = 6`

`bits = "1"`

`rem = 1`

OUTPUT

```
13 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
▶    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```

n	÷ 2	rem
13	→ 6	1

bits: 1

`6 > 0` is true → enter the loop.

Step 6 / 16

VARIABLES

- `n = 6`
- `bits = "1"`
- `rem = 1`

OUTPUT

```
13 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
▶      int rem = n % 2;
▶      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```

n	÷ 2	rem
13	→ 6	1
6	→ 3	0

bits `1`

Remainder `6 % 2 = 0` : the **rightmost** bit not yet written.

Step 7 / 16

VARIABLES

`n = 6`

`bits = "1"`

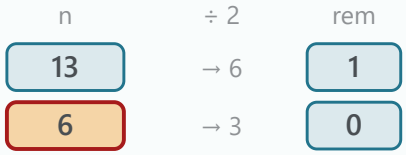
`rem = 0`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
▶      bits = rem + bits;
▶      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```





The remainder goes in **front** of the string → "01"; n is halved: 6 / 2 = 3 (integer division).

Step 8 / 16

VARIABLES

`n = 3`

`bits = "01"`

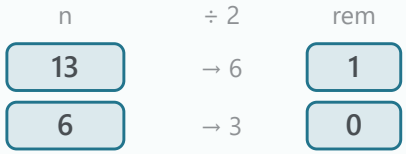
`rem = 0`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
▶    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



bits 0 1

`3 > 0` is true → enter the loop.

Step 9 / 16

VARIABLES

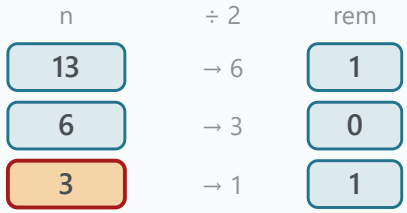
- `n = 3`
- `bits = "01"`
- `rem = 0`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
▶      int rem = n % 2;
▶      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



bits **0** **1**

Remainder `3 % 2 = 1` : the **rightmost** bit not yet written.

Step 10 / 16

VARIABLES

`n = 3`

`bits = "01"`

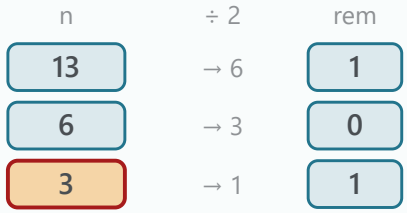
`rem = 1`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
▶      bits = rem + bits;
▶      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```





The remainder goes in **front** of the string → "101"; n is halved: 3 / 2 = 1 (integer division).

Step 11 / 16

VARIABLES

`n = 1`

`bits = "101"`

`rem = 1`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
▶    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```

n	÷ 2	rem
13	→ 6	1
6	→ 3	0
3	→ 1	1

bits: 1 0 1

`1 > 0` is true → enter the loop.

Step 12 / 16

VARIABLES

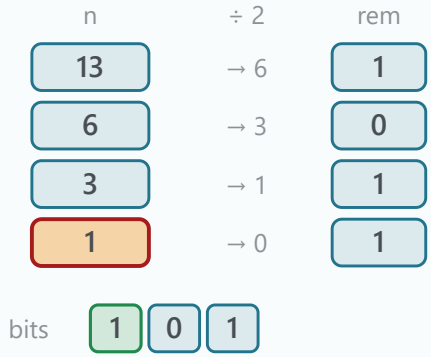
- `n = 1`
- `bits = "101"`
- `rem = 1`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
▶      int rem = n % 2;
▶      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



Remainder `1 % 2 = 1` : the **rightmost** bit not yet written.

Step 13 / 16

VARIABLES

`n = 1`

`bits = "101"`

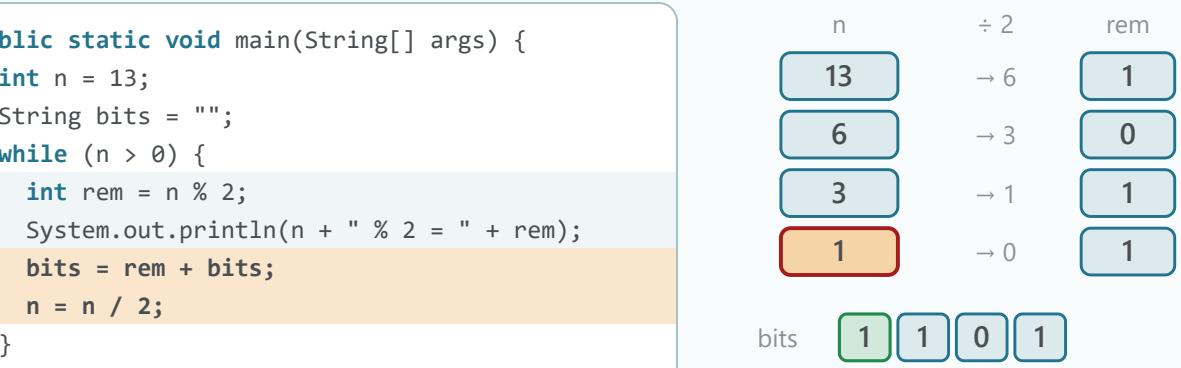
`rem = 1`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
1 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
▶      bits = rem + bits;
▶      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



The remainder goes in **front** of the string → "1101"; n is halved: 1 / 2 = 0 (integer division).

Step 14 / 16

VARIABLES

`n = 0`

`bits = "1101"`

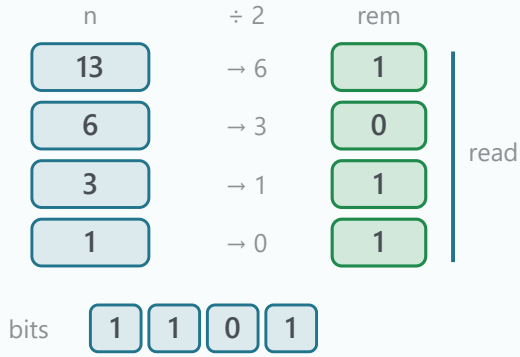
`rem = 1`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
1 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
   ▶ while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
10   System.out.println("Result: " + bits);
11 }
```



`0 > 0` is false → the loop ends. The remainders are read **bottom to top**.

Step 15 / 16

VARIABLES

`n = 0`

`bits = "1101"`

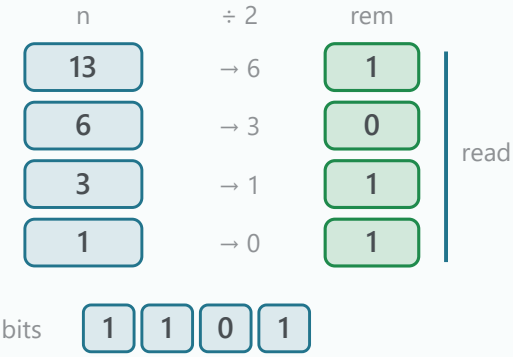
`rem = –`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
1 % 2 = 1
```

Decimal to Binary: Repeated Division by 2

```
1  public static void main(String[] args) {
2    int n = 13;
3    String bits = "";
4    while (n > 0) {
5      int rem = n % 2;
6      System.out.println(n + " % 2 = " + rem);
7      bits = rem + bits;
8      n = n / 2;
9    }
▶    System.out.println("Result: " + bits);
11 }
```



The result is printed: 13_{10} = **1101_2**.

Step 16 / 16

VARIABLES

`n = 0`

`bits = "1101"`

`rem = –`

OUTPUT

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
1 % 2 = 1
Result: 1101
```

NUMBERS

Base Conversion in Java

```
System.out.println(Integer.toBinaryString(13));  // binary
System.out.println(Integer.toOctalString(135));  // octal
System.out.println(Integer.toHexString(255));    // hex
System.out.println(Integer.parseInt("1101", 2)); // 2 → 10
System.out.println(Integer.parseInt("207", 8));  // 8 → 10
System.out.println(0b1101 + " " + 0207 + " " + 0xFF);
```

Output:

```
1101
207
ff
13
135
13 135 255
```

`0b` prefix

Binary literal: `0b1101` = 13 (Java 7+).

`0` prefix

Octal literal: `0207` = 135. `010` is **8**, not ten!

`0x` prefix

Hex literal: `0xFF` = 255. Common for colors and byte values.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.



Write the two numbers one above the other: 57 = 00111001_2 and 28 = 00011100_2. Addition starts at the **right**.

Step 1 / 9

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.



Column 1: 1 + 0 = 1 → write 1, no carry.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

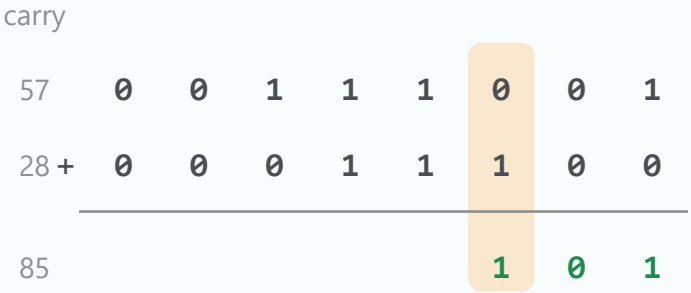


Column 2: 0 + 0 = 0 → write 0, no carry.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```



In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

Column 3: 0 + 1 = 1 → write 1, no carry.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

carry				1				
57	0	0	1	1	1	0	0	1
28 +	0	0	0	1	1	1	0	0
85					0	1	0	1

Column 4: 1 + 1 = 2 = **10_2** → write 0, **carry 1**.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

carry			1	1				
57	0	0	1	1	1	0	0	1
28 +	0	0	0	1	1	1	0	0
85				1	0	1	0	1

Column 5: 1 + 1 + 1 (carry) = 3 = $\mathbf{11_2}$ → write 1, **carry 1**.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

carry		1	1	1				
57	0	0	1	1	1	0	0	1
28 +	0	0	0	1	1	1	0	0
85			0	1	0	1	0	1

Column 6: 1 + 0 + 1 (carry) = 2 = **10_2** → write 0, **carry 1**.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

carry		1	1	1				
57	0	0	1	1	1	0	0	1
28 +	0	0	0	1	1	1	0	0
85		1	0	1	0	1	0	1

Column 7: 0 + 0 + 1 (carry) = 1 → write 1, no carry.

Addition Follows the Same Rules in Every Base

Whatever the base, the same principles apply: **place values**, the **carry**, and a correct result.

```
   1
  57
+ 28
----
  85
```

In decimal 7 + 8 = 15 → write 5, carry 1. In binary 1 + 1 = 2 = 10_2 → write 0, carry 1.

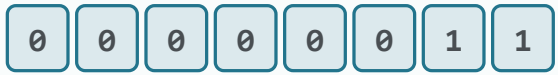
carry		1	1	1				
57	0	0	1	1	1	0	0	1
28 +	0	0	0	1	1	1	0	0
85	0	1	0	1	0	1	0	1

Column 8: 0 + 0 = 0 → write 0, no carry. Result: 01010101_2 = 64 + 16 + 4 + 1 = **85** ✓

Negative Numbers: Two's Complement

- Negative integers are usually stored in **two's complement**.
- Rule: **invert the bits** of the positive number, then **add 1**.
- The leftmost bit acts as a **sign bit**: 0 → positive, 1 → negative.
- Range with 8 bits: **−128 … 127**. With n bits: $-2^{n-1} \dots 2^{n-1} - 1$.

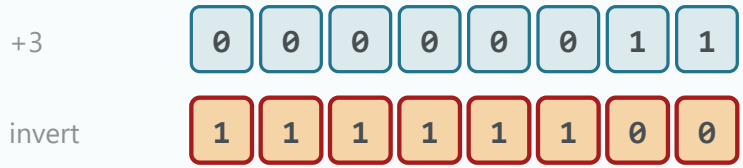
+3



On an 8-bit computer, 3 is **00000011**.

Negative Numbers: Two's Complement

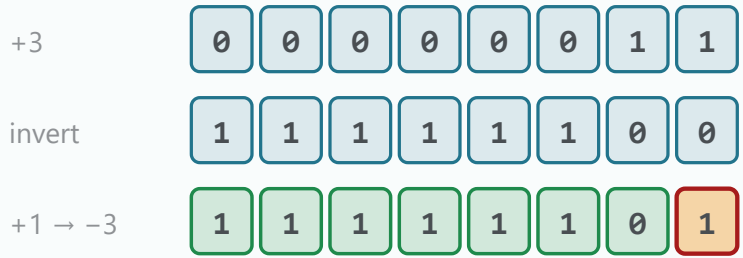
- Negative integers are usually stored in **two's complement**.
- Rule: **invert the bits** of the positive number, then **add 1**.
- The leftmost bit acts as a **sign bit**: 0 → positive, 1 → negative.
- Range with 8 bits: **−128 … 127**. With n bits: $-2^{n-1} \dots 2^{n-1} - 1$.



Step 1: **invert** every bit (0 ↔ 1): 00000011 → 11111100 (the ones' complement).

Negative Numbers: Two's Complement

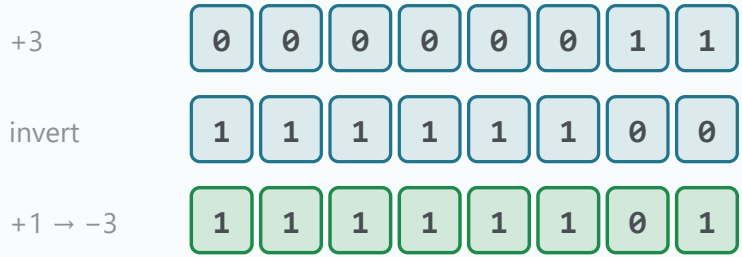
- Negative integers are usually stored in **two's complement**.
- Rule: **invert the bits** of the positive number, then **add 1**.
- The leftmost bit acts as a **sign bit**: 0 → positive, 1 → negative.
- Range with 8 bits: **−128 … 127**. With n bits: $-2^{n-1} \dots 2^{n-1} - 1$.



Step 2: **add 1**: 11111100 + 1 → **11111101**. That is −3.

Negative Numbers: Two's Complement

- Negative integers are usually stored in **two's complement**.
- Rule: **invert the bits** of the positive number, then **add 1**.
- The leftmost bit acts as a **sign bit**: 0 → positive, 1 → negative.
- Range with 8 bits: **−128 … 127**. With n bits: $-2^{n-1} \dots 2^{n-1} - 1$.

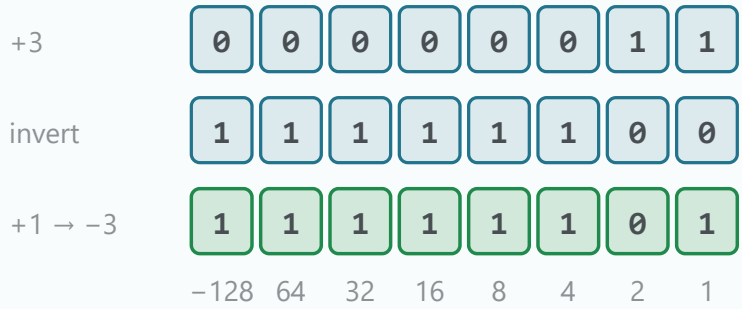


Check: 00000011 + 11111101 = 1 00000000
9th bit doesn't fit, dropped → 00000000 = 0 ✓

Why does it work? Adding 3 + (−3), the 9th bit overflows and is dropped, leaving exactly 0: the ordinary adder does the job.

Negative Numbers: Two's Complement

- Negative integers are usually stored in **two's complement**.
- Rule: **invert the bits** of the positive number, then **add 1**.
- The leftmost bit acts as a **sign bit**: 0 → positive, 1 → negative.
- Range with 8 bits: **−128 … 127**. With n bits: $-2^{n-1} \dots 2^{n-1} - 1$.



−128 + 64 + 32 + 16 + 8 + 4 + 1 = −3

Another view: the leftmost bit has weight **−128**, so when it is 1 the number is negative.

Two's Complement and Overflow in Java

```java
int n = 3;
int neg = ~n + 1; // invert, add 1
System.out.println(neg);
byte b = (byte) 0b11111101;
System.out.println(b);
System.out.println(Integer.toBinaryString(-3));
System.out.println(Byte.MIN_VALUE + " .. "
    + Byte.MAX_VALUE);
```

```
-3
-3
11111111111111111111111111111101
-128 .. 127
```

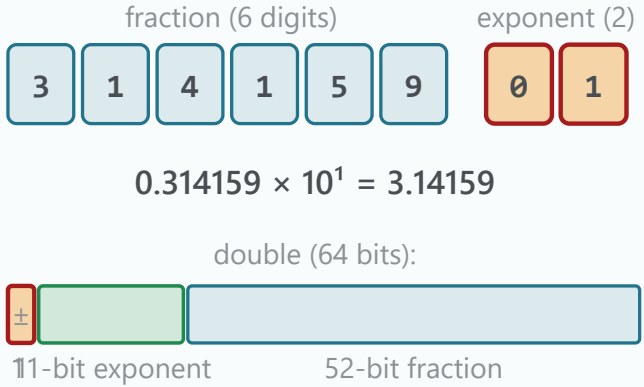
```java
byte b = 127;
b++;
System.out.println(b);
int m = Integer.MAX_VALUE;
System.out.println(m + 1);
System.out.println(Math.addExact(m, 1));
```

```
-128
-2147483648
Exception in thread "main"
  java.lang.ArithmeticException:
  integer overflow
```

Overflow: when a result doesn't fit the type's range, Java **reports no error**; the value wraps around like a clock: 127 + 1 → −128. Use `Math.addExact` for checked addition.

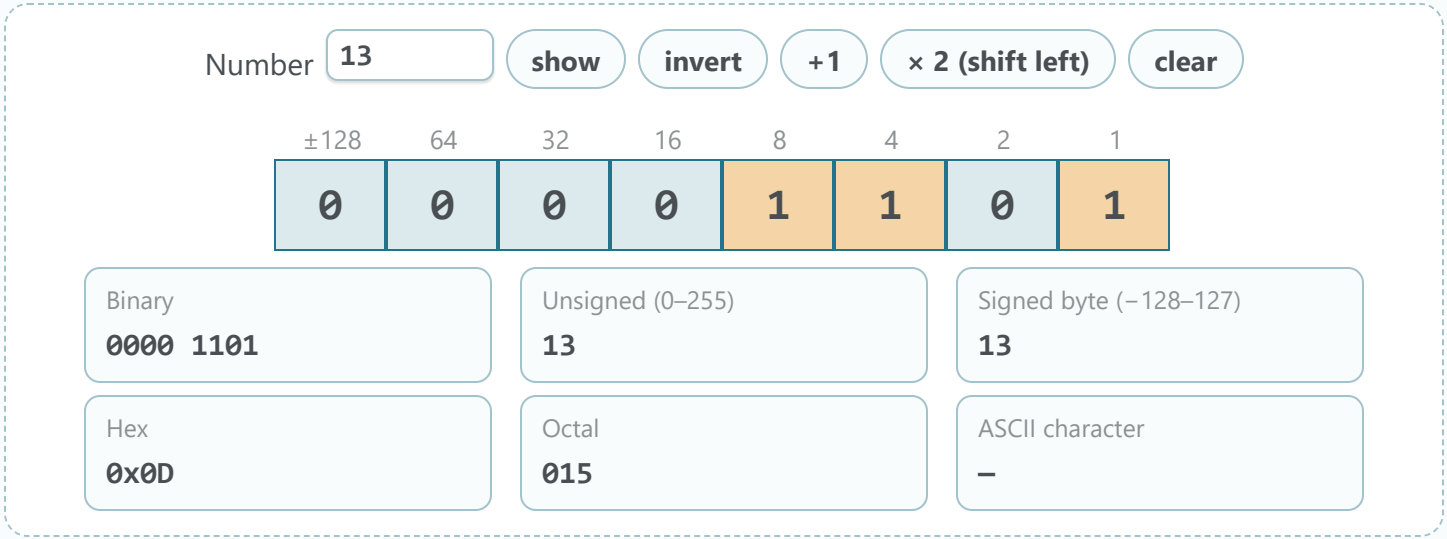
Very Large and Very Small Numbers: Floating Point

- A fixed number of digits can represent only a fixed number of values.
- For very large or small numbers, the value is split into a **fraction** (mantissa) and an **exponent**, stored separately.
- Example: in the 8-digit form **31415901**, the first six digits (314159) are the fraction and the last two (01) the power of 10, i.e. 10^1.
- Computers apply the same idea in **binary**: the **IEEE 754** standard. In Java, `float` is 32 bits and `double` 64.



Hence the name *floating point*: as the exponent changes, the decimal point moves: $0.314159 \times 10^3 = 314.159$.

Playing with Bits



Clicking a bit flips it 0 ↔ 1. For −3: 3 → **invert** → **+1**. The signed value counts the leftmost bit as −128.

Letters as Numbers: ASCII

- To process text, every letter must be mapped to a **number**: a **character encoding**.
- **ASCII** (*American Standard Code for Information Interchange*) was published in 1963 by the US standards body ASA (today **ANSI**).
- It assigns numbers to upper- and lowercase letters, digits, punctuation and special characters.
- It is a **7-bit** code: **128** characters (0–127). 0–31 and 127 are control characters; 32–126 are printable.
- **8-bit** extensions (e.g. ISO 8859-9 and Windows-1254 for Turkish) put extra letters in 128–255, for **256** symbols.
- No single 8-bit table is **enough** for all the world's languages.

Character	Decimal	Binary
`'A'`	65	`1000001`
`'a'`	97	`1100001`
`'0'`	48	`0110000`
space	32	`0100000`
newline `\n`	10	`0001010`

Lowercase = uppercase + **32**. In binary they differ in a single bit (2^5).

The ASCII Table: Printable Characters

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	SP 32	! 33	" 34	# 35	$ 36	% 37	& 38	' 39	(40	) 41	* 42	+ 43	, 44	- 45	. 46	/ 47
48	0 48	1 49	2 50	3 51	4 52	5 53	6 54	7 55	8 56	9 57	: 58	; 59	< 60	= 61	> 62	? 63
64	@ 64	A 65	B 66	C 67	D 68	E 69	F 70	G 71	H 72	I 73	J 74	K 75	L 76	M 77	N 78	O 79
80	P 80	Q 81	R 82	S 83	T 84	U 85	V 86	W 87	X 88	Y 89	Z 90	[91	\ 92	] 93	^ 94	_ 95
96	` 96	a 97	b 98	c 99	d 100	e 101	f 102	g 103	h 104	i 105	j 106	k 107	l 108	m 109	n 110	o 111
112	p 112	q 113	r 114	s 115	t 116	u 117	v 118	w 119	x 120	y 121	z 122	{ 123	| 124	} 125	~ 126	DEL 127

Code = row + column. Example: `'J'` = 64 + 10 = **74**. Each cell shows the character above its decimal code.

0–31 are control characters, e.g. 9 tab (`\t`), 10 newline (`\n`), 13 carriage return (`\r`).

One Table for Every Language: Unicode

- **Unicode** gives every letter, symbol and mark of the world's writing systems a number, its **code point**: `U+0041` = A, `U+00E7` = ç.
- It is developed by the **Unicode Consortium** (1991) and kept in sync with the international standard **ISO/IEC 10646**.
- The first version was 16-bit (65,536 characters). When that ran out, the range grew to **U+10FFFF**: 1,114,112 code points, over 150,000 characters today.
- Storing every character in a fixed 32 bits (UTF-32) wastes space on text like English. Hence the **UTF** (*Unicode Transformation Format*) encodings.

Char	Code point	Decimal	UTF-8 bytes
A	`U+0041`	65	`41`
ç	`U+00E7`	231	`C3 A7`
ğ	`U+011F`	287	`C4 9F`
€	`U+20AC`	8364	`E2 82 AC`
😀	`U+1F600`	128512	`F0 9F 98 80`

The first 128 code points are identical to ASCII.

UTF-8: A Variable-Length Encoding

- Common characters take **few** bytes, rare ones **more**: 1–4 bytes.
- ASCII text is valid UTF-8 as is (backward compatible). The vast majority of web pages today are UTF-8.
- The leading bits of the first byte tell how many bytes the character spans.
- **UTF-16** writes most characters in 2 bytes (`ş` = `01 5F` , `€` = `20 AC`); Java's `char` is one UTF-16 unit.

Code point	Byte pattern
U+0000–U+007F	`0xxxxxxx`
U+0080–U+07FF	`110xxxxx 10xxxxxx`
U+0800–U+FFFF	`1110xxxx 10xxxxxx ×2`
U+10000–U+10FFFF	`11110xxx 10xxxxxx ×3`

U+00E7 = 231 = 00011 100111 (11 bits)



colored: pattern bits · others: code point bits

```
String word = "çikolata";
System.out.println(word.length());
System.out.println(word.getBytes(
    StandardCharsets.UTF_8).length);
```

Output:

```
8
9
```

An Emoji Is a Character Too

- Emoji are symbols that quickly convey feelings, expressions and objects; the computer treats them as **text**.
- The **Unicode Consortium** sets the standard: each emoji is a unique code point, e.g. 😀 = `U+1F600`.
- Some emoji combine **several code points**: 👨‍👩‍👧 = man + ZWJ + woman + ZWJ + girl (5 code points; ZWJ = zero-width joiner).
- The code is the **same** on every device, so the meaning survives; the **drawing** (glyph) may differ by vendor.

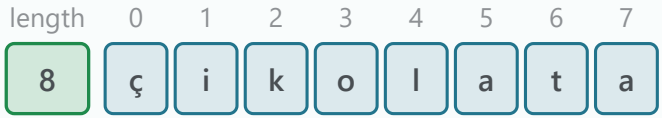
```
String face = "😀";
System.out.println(face.length());
System.out.println(
    face.codePointCount(0, face.length()));
String family = "👨‍👩‍👧";
System.out.println(family.length());
```

Output:

```
2
1
8
```

A Java `String` stores text in **UTF-16**: a `char` is 16 bits, and characters above U+FFFF take **two** `char`s (a *surrogate pair*). So `length()` counts `char`s, not characters.

Printing a String's Character Codes



In memory, a string is an array of characters plus a length.

Step 1 / 11

```
String word = "çikolata";
for (int i = 0; i < word.length(); i++) {
  char ch = word.charAt(i);
  System.out.print((int) ch + " ");
}
System.out.println();
System.out.println("Length: " + word.length());
```

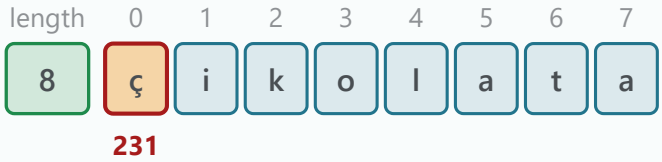
VARIABLES

word = "çikolata" i = – ch = –

OUTPUT

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 0: `charAt(0)` = 'ç'; the `(int)` cast gives its code: **231** (Unicode U+00E7).

Step 2 / 11

```
1 String word = "çikolata";
▶ for (int i = 0; i < word.length(); i++) {
▶   char ch = word.charAt(i);
▶   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

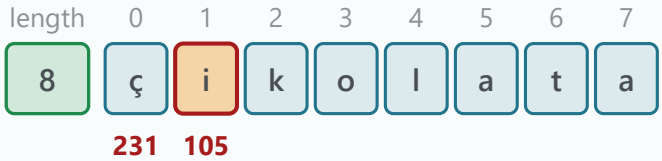
`word = "çikolata"` `i = 0` `ch = 'ç'`

OUTPUT

```
231
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 1: `charAt(1)` = 'i'; the `(int)` cast gives its code: **105**.

Step 3 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);
    System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

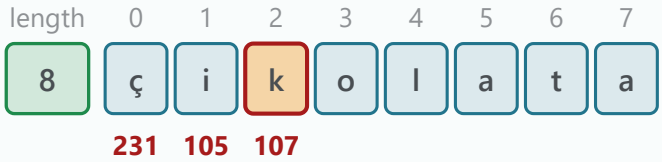
`word = "çikolata"` `i = 1` `ch = 'i'`

OUTPUT

```
231 105
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 2: `charAt(2)` = 'k'; the `(int)` cast gives its code: **107**.

Step 4 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
    char ch = word.charAt(i);
    System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

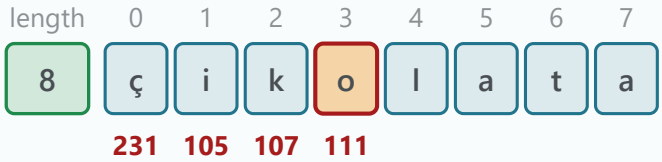
`word = "çikolata"` `i = 2` `ch = 'k'`

OUTPUT

```
231 105 107
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 3: `charAt(3)` = 'o'; the `(int)` cast gives its code: **111**.

Step 5 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
▶   char ch = word.charAt(i);
▶   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

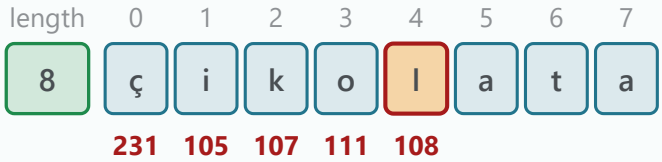
`word = "çikolata"` `i = 3` `ch = 'o'`

OUTPUT

```
231 105 107 111
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 4: `charAt(4)` = 'l'; the `(int)` cast gives its code: **108**.

Step 6 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
▶   char ch = word.charAt(i);
▶   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

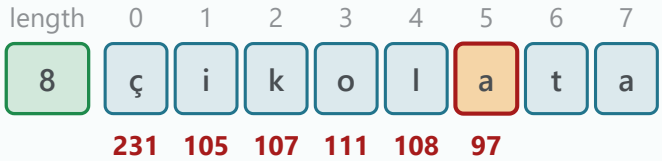
`word = "çikolata"` `i = 4` `ch = 'l'`

OUTPUT

```
231 105 107 111 108
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 5: `charAt(5)` = 'a'; the `(int)` cast gives its code: **97**.

Step 7 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
▶   char ch = word.charAt(i);
▶   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

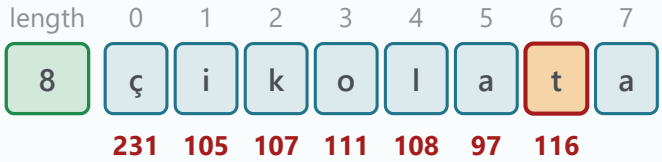
`word = "çikolata"` `i = 5` `ch = 'a'`

OUTPUT

```
231 105 107 111 108 97
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 6: `charAt(6)` = 't'; the `(int)` cast gives its code: **116**.

Step 8 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
▶   char ch = word.charAt(i);
▶   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

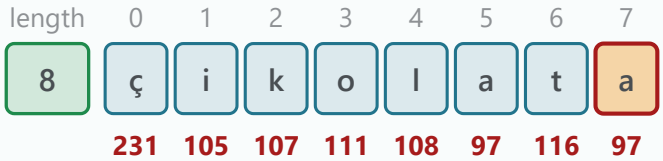
`word = "çikolata"` `i = 6` `ch = 't'`

OUTPUT

```
231 105 107 111 108 97 116
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 7: `charAt(7)` = 'a'; the `(int)` cast gives its code: **97**.

Step 9 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
▶   char ch = word.charAt(i);
▶   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

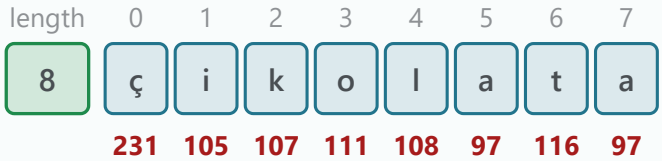
`word = "çikolata"` `i = 7` `ch = 'a'`

OUTPUT

```
231 105 107 111 108 97 116 97
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



i = 8: `8 < 8` is false → the loop ends.

Step 10 / 11

```
1 String word = "çikolata";
▶ for (int i = 0; i < word.length(); i++) {
3   char ch = word.charAt(i);
4   System.out.print((int) ch + " ");
5 }
6 System.out.println();
7 System.out.println("Length: " + word.length());
```

VARIABLES

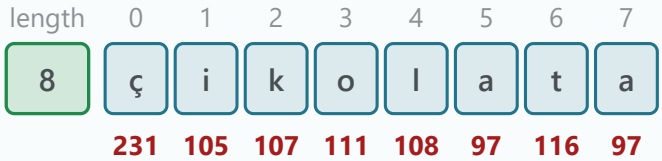
`word = "çikolata"` `i = 8` `ch = –`

OUTPUT

```
231 105 107 111 108 97 116 97
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Printing a String's Character Codes



A newline and the length are printed: `length()` = 8.

Step 11 / 11

```
1 String word = "çikolata";
2 for (int i = 0; i < word.length(); i++) {
3   char ch = word.charAt(i);
4   System.out.print((int) ch + " ");
5 }
▶ System.out.println();
▶ System.out.println("Length: " + word.length());
```

VARIABLES

`word = "çikolata"` `i = –` `ch = –`

OUTPUT

```
231 105 107 111 108 97 116 97
Length: 8
```

The length is stored separately; that is how we know where the text ends. “ç” isn't in ASCII: 231 is its Unicode code point (U+00E7), and it is also 231 in Latin-1 and the Turkish ISO 8859-9.

Playing with Character Codes



ç	i	k	o	l	a
231	105	107	111	108	97
U+00E7	U+0069	U+006B	U+006F	U+006C	U+0061
C3 A7	69	6B	6F	6C	61

t	a	SP	😀
116	97	32	128512
U+0074	U+0061	U+0020	U+1F600
74	61	20	F0 9F 98 80

Code points: **10** · Java `length()` (UTF-16 char): **11** · UTF-8 bytes: **14**

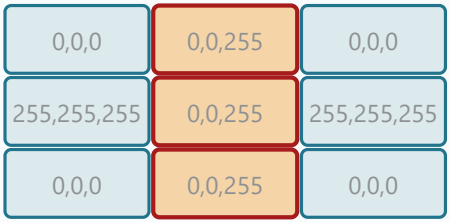
Each box: character · code point (decimal and U+hex) · UTF-8 bytes. Colored boxes are non-ASCII. The first 12 code points are shown.

STRUCTURED INFORMATION

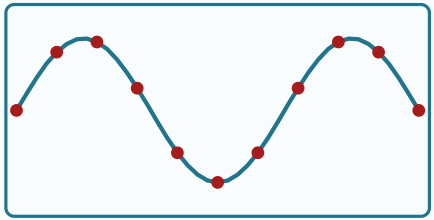
Every Kind of Information Is Numbers

Sequences of numbers let us store information and interact with the computer.

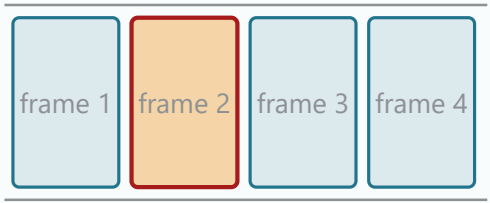
Image: pixel = (R, G, B)



Sound: pressure samples



Film: 24–30 frames per second



Images

Each pixel is **three numbers** (0–255): red, green and blue levels. A blue pixel: (0, 0, 255).

Sound

Air pressure is **measured** at regular intervals (sampling); CD quality is 44,100 samples per second.

Film

A run of **images** at 24 or 30 frames per second; each frame is a sequence of numbers, like any image.

Example: How Much Space Does a Video Take?

```java
long pixels = 1920L * 1080;  // one Full HD frame
long frame = pixels * 3;      // R, G, B: 3 bytes
long perSecond = frame * 30;     // 30 frames/s
System.out.println("Pixels: " + pixels);
System.out.println("Frame (bytes): " + frame);
System.out.println("1 second (bytes): " + perSecond);
System.out.println("1 minute (MB): "
    + perSecond * 60 / 1_000_000);
// 44.1 kHz, 16 bits (2 bytes), stereo, 1 min
long audio = 44_100L * 2 * 2 * 60;
System.out.println("1 min CD audio (bytes): " + audio);
```

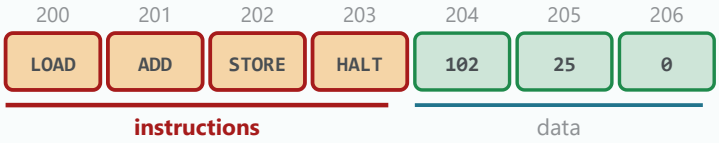
Output:

```
Pixels: 2073600
Frame (bytes): 6220800
1 second (bytes): 186624000
1 minute (MB): 11197
1 min CD audio (bytes): 10584000
```

One minute of uncompressed Full HD video ≈ **11 GB**! In practice video and audio are stored **compressed** (H.264, MP3…), but a compressed file is still a sequence of numbers. Note: `1920L` makes the product a `long`; large results don't fit in an `int`.

The Stored-Program Concept

- The core principle of modern computers: **instructions are numbers, just like data**, and live in the same memory.
- Together, the instructions and data in memory form the **program**. Each instruction stands for one task or operation.
- Instructions are executed **sequentially**.
- When a condition holds, the program can **jump** to a different instruction, so it can take different paths (`if` and loops build on this later).



ENIAC (1945)

- Programs are set up **by hand** with cables and switches.
- A new program means days of rewiring.

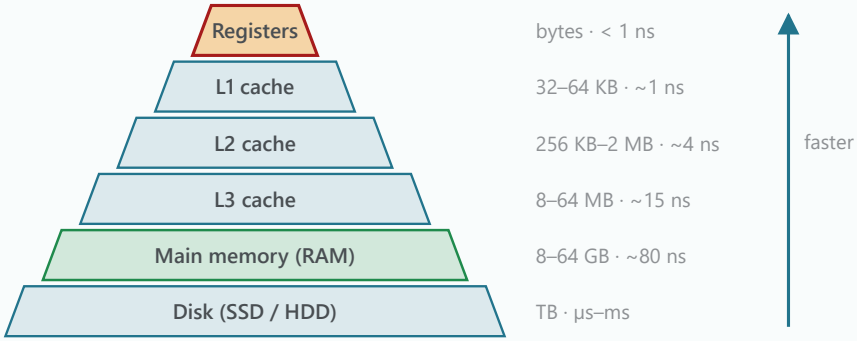
EDVAC design (1945)

+ Instructions and data are stored **in sequence** in memory.
+ Changing the program = writing new numbers to memory: flexible and extensible.

J. von Neumann's 1945 EDVAC report spread the idea; the first machine to run a stored program was the Manchester “Baby” (1948).

The Memory Hierarchy

- **Memory** temporarily holds running programs and their data.
- **L1 cache**: the fastest, inside the processor.
- **L2 cache**: backs up L1 and is larger. Most processors also have an **L3**.
- **Main memory** (RAM): much larger but slower.
- Memory close to the processor is faster but smaller and costlier; the hierarchy directly affects performance.



Sizes and times are approximate and vary by processor. 1 ns = one billionth of a second.

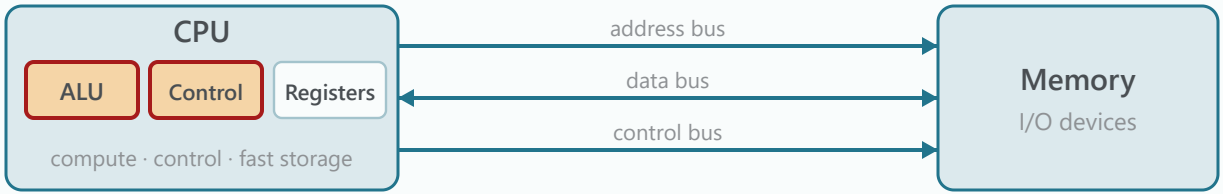
The Processor (CPU) and the Bus

Central Processing Unit (CPU)

- The “brain” of the computer; all computation happens here.
- **Arithmetic logic unit** (ALU): does addition, subtraction, comparison and the like.
- **Control unit**: decodes instructions and sequences operations.
- **Registers**: tiny but very fast storage for temporary values.

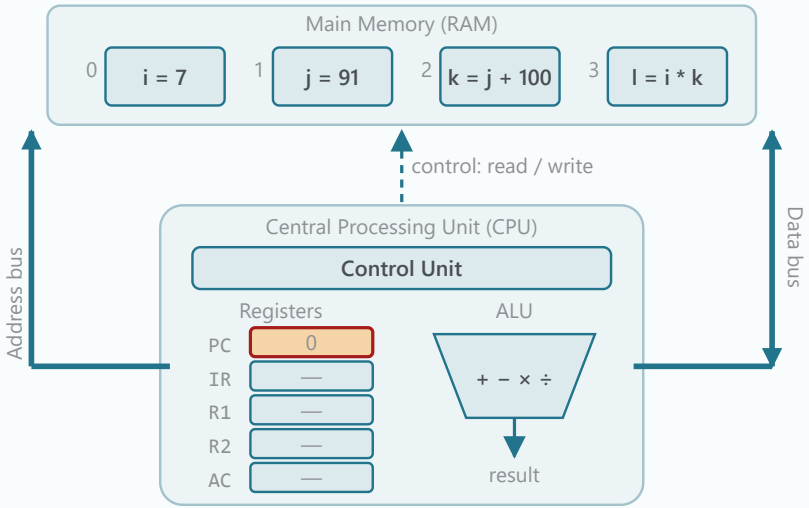
The Bus

- A **communication pathway** made of many parallel wires.
- **Address signals** select which memory cell or device to access.
- **Data signals** carry the actual data to process or store.
- **Control signals** direct the operation, e.g. “read” or “write”.



The electronics link: a 32-line address bus can select 2^{32} addresses → 4 GiB of memory. That is why 32-bit systems could use at most about 4 GB of RAM.

CPU – Memory – Bus – Registers



- The CPU names the cell it wants on the **address bus**.
- The cell's contents travel over the **data bus**: in (read) or out (write).
- **PC** (program counter): address of the next instruction.
- **IR** (instruction register): the instruction being decoded.
- **R1, R2, AC** (accumulator): intermediate results.
- The ALU computes with the numbers in registers.

STORED PROGRAM

Memory, Addresses and the Program Counter

- Memory holds data and instructions; a program is **a long list of instructions**.
- Every memory cell has an **address** and holds an instruction or a piece of data **encoded as a number**.
- Instructions are fetched one at a time into the processor's **registers** and executed in order.
- One of the most important registers is the **program counter** (PC): it holds the address of the next instruction.
- Normally the processor moves through addresses in order, but some instructions **change the order**: "JUMP 7" at 204 sets the PC to 211.

Address	Contents	Kind
200	`ADD to R1`	instruction
201	`102`	data
202	…	
203	…	
204	`JUMP 7`	instruction (jump)
…	…	
211	next instruction	instruction

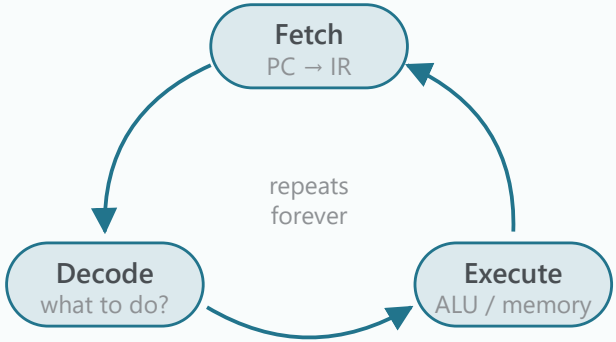
You can't tell from a cell whether it holds an instruction or data: both are numbers. The difference is **how the processor uses it**.

The Fetch–Decode–Execute Cycle

The **fetch–decode–execute** cycle describes how the processor runs a program.

1. **Fetch**: the processor reads the next instruction from the address in the PC; the PC advances to the next address.
2. **Decode**: the instruction is decoded: which operation, which registers, which address? Each instruction has a specific function.
3. **Execute**: the operation is carried out: processing data, computing a result or starting an action.

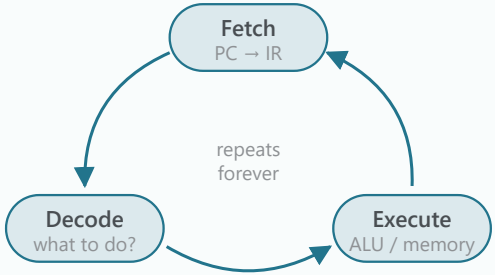
Then the next instruction is fetched, and the cycle repeats **continuously** as long as the computer is on.



A 3 GHz processor's clock “ticks” 3 billion times a second; modern processors overlap several instructions (a *pipeline*).

102 + 25 on a Toy Computer

```
▶ 200: LOAD  R1, [205] ; R1 ← mem[205]
  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
  202: STORE R1, [207] ; mem[207] ← R1
  203: HALT            ; stop
  204: ...
  205: 102             ; data
  206: 25              ; data
  207: 0               ; result here
```



Program and data are in memory. Initially **PC = 200**.

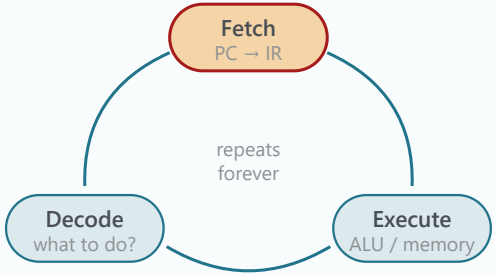
Step 1 / 12

VARIABLES

`PC = 200` `IR = –` `R1 = 0` `mem[207] = 0`

102 + 25 on a Toy Computer

```
▶ 200: LOAD  R1, [205] ; R1 ← mem[205]
  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
  202: STORE R1, [207] ; mem[207] ← R1
  203: HALT            ; stop
  204: ...
  205: 102             ; data
  206: 25              ; data
  207: 0               ; result here
```



Fetch: the instruction at 200 goes into IR; PC advances to 201.

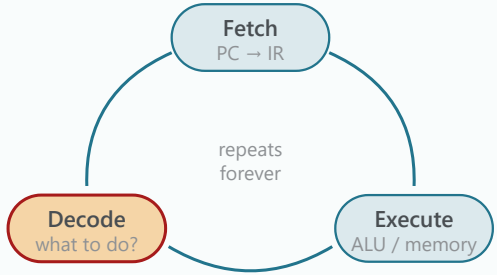
Step 2 / 12

VARIABLES

`PC = 201` `IR = LOAD R1,[205]` `R1 = 0` `mem[207] = 0`

102 + 25 on a Toy Computer

```
▶ 200: LOAD  R1, [205] ; R1 ← mem[205]
  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
  202: STORE R1, [207] ; mem[207] ← R1
  203: HALT            ; stop
  204: ...
  205: 102             ; data
  206: 25              ; data
  207: 0               ; result here
```



Decode: operation = LOAD, target = R1, address = 205.

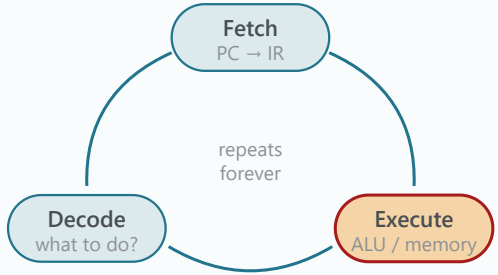
Step 3 / 12

VARIABLES

`PC = 201` `IR = LOAD R1,[205]` `R1 = 0` `mem[207] = 0`

102 + 25 on a Toy Computer

```
▶ 200: LOAD  R1, [205] ; R1 ← mem[205]
  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
  202: STORE R1, [207] ; mem[207] ← R1
  203: HALT            ; stop
  204: ...
▶ 205: 102             ; data
  206: 25              ; data
  207: 0               ; result here
```



Execute: mem[205] = 102 is read into R1.

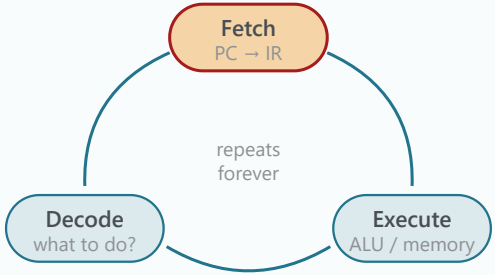
Step 4 / 12

VARIABLES

PC = **201** | IR = **LOAD R1,[205]** | R1 = **102** | mem[207] = **0**

102 + 25 on a Toy Computer

```
1  200: LOAD  R1, [205] ; R1 ← mem[205]
▶  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
3  202: STORE R1, [207] ; mem[207] ← R1
4  203: HALT            ; stop
5  204: ...
6  205: 102             ; data
7  206: 25              ; data
8  207: 0               ; result here
```



Fetch: the instruction at PC = 201 is read; PC becomes 202.

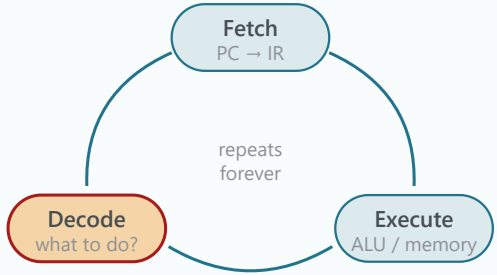
Step 5 / 12

VARIABLES

PC = 202 | IR = ADD R1,[206] | R1 = 102 | mem[207] = 0

102 + 25 on a Toy Computer

```
1  200: LOAD  R1, [205] ; R1 ← mem[205]
▶  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
3  202: STORE R1, [207] ; mem[207] ← R1
4  203: HALT            ; stop
5  204: ...
6  205: 102             ; data
7  206: 25              ; data
8  207: 0               ; result here
```



Decode: operation = ADD: add R1 and mem[206].

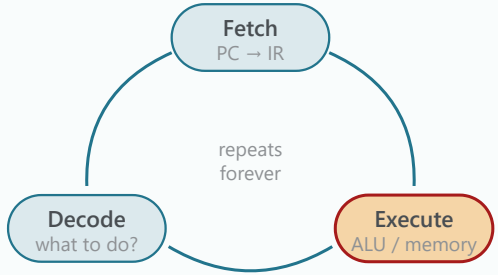
Step 6 / 12

VARIABLES

`PC = 202` `IR = ADD R1,[206]` `R1 = 102` `mem[207] = 0`

102 + 25 on a Toy Computer

```
1  200: LOAD  R1, [205] ; R1 ← mem[205]
▶  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
3  202: STORE R1, [207] ; mem[207] ← R1
4  203: HALT            ; stop
5  204: ...
6  205: 102             ; data
▶  206: 25              ; data
8  207: 0               ; result here
```



Execute: the ALU computes 102 + 25 = **127** into R1.

Step 7 / 12

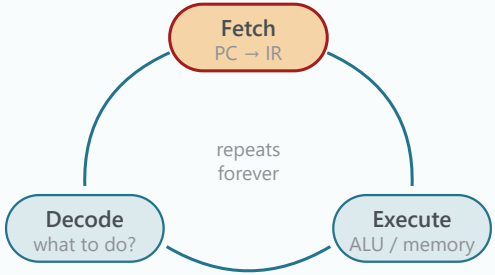
VARIABLES

`PC = 202` `IR = ADD R1,[206]`

`R1 = 127` `mem[207] = 0`

102 + 25 on a Toy Computer

```
1 200: LOAD  R1, [205] ; R1 ← mem[205]
2 201: ADD   R1, [206] ; R1 ← R1 + mem[206]
▶ 202: STORE R1, [207] ; mem[207] ← R1
4 203: HALT            ; stop
5 204: ...
6 205: 102             ; data
7 206: 25              ; data
8 207: 0               ; result here
```



Fetch: the STORE at 202 is read; PC becomes 203.

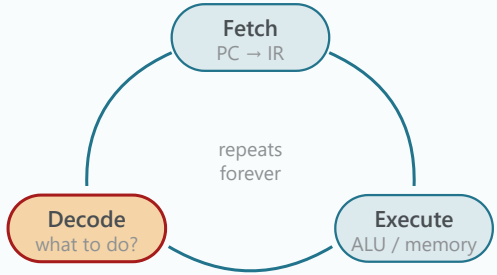
Step 8 / 12

VARIABLES

PC = 203 | IR = STORE R1,[207] | R1 = 127 | mem[207] = 0

102 + 25 on a Toy Computer

```
1 200: LOAD  R1, [205] ; R1 ← mem[205]
2 201: ADD   R1, [206] ; R1 ← R1 + mem[206]
▶ 202: STORE R1, [207] ; mem[207] ← R1
4 203: HALT            ; stop
5 204: ...
6 205: 102             ; data
7 206: 25              ; data
8 207: 0               ; result here
```



Decode: R1 is to be written to address 207 (control bus: “write”).

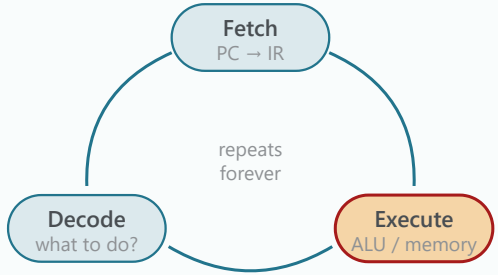
Step 9 / 12

VARIABLES

`PC = 203` `IR = STORE R1,[207]` `R1 = 127` `mem[207] = 0`

102 + 25 on a Toy Computer

```
1 200: LOAD  R1, [205] ; R1 ← mem[205]
2 201: ADD   R1, [206] ; R1 ← R1 + mem[206]
▶ 202: STORE R1, [207] ; mem[207] ← R1
4 203: HALT            ; stop
5 204: ...
6 205: 102             ; data
7 206: 25              ; data
▶ 207: 0               ; result here
```



Execute: mem[207] ← 127. The result is now in memory.

Step 10 / 12

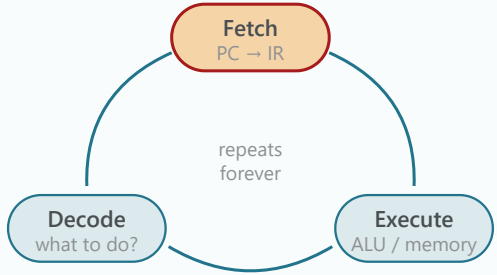
VARIABLES

PC = 203 | IR = STORE R1,[207]

R1 = 127 | mem[207] = 127

102 + 25 on a Toy Computer

```
1  200: LOAD  R1, [205] ; R1 ← mem[205]
2  201: ADD   R1, [206] ; R1 ← R1 + mem[206]
3  202: STORE R1, [207] ; mem[207] ← R1
▶  203: HALT            ; stop
5  204: ...
6  205: 102             ; data
7  206: 25              ; data
8  207: 0               ; result here
```



Fetch: the HALT at 203 is read; PC becomes 204.

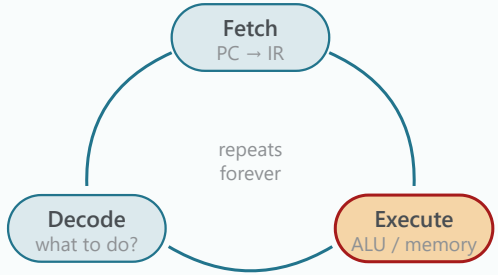
Step 11 / 12

VARIABLES

`PC = 204` `IR = HALT` `R1 = 127` `mem[207] = 127`

102 + 25 on a Toy Computer

```
1 200: LOAD  R1, [205] ; R1 ← mem[205]
2 201: ADD   R1, [206] ; R1 ← R1 + mem[206]
3 202: STORE R1, [207] ; mem[207] ← R1
▶ 203: HALT            ; stop
5 204: ...
6 205: 102             ; data
7 206: 25              ; data
8 207: 0               ; result here
```



Decode + Execute: HALT → the processor stops. A Java line like `sum = a + b;` ends up as instructions like these.

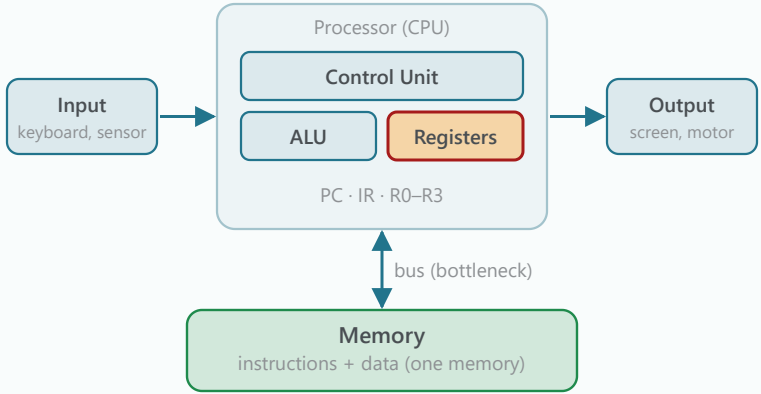
Step 12 / 12

VARIABLES

`PC = 204` `IR = HALT` `R1 = 127` `mem[207] = 127`

The von Neumann Architecture

- The hardware blueprint of the stored-program idea: **processor**, **memory**, **input/output** and the **bus** that connects them.
- Instructions and data sit in **the same memory** and travel over the same bus.
- Each instruction carries an **opcode** that says which operation to perform.
- A single bus can leave the processor waiting: the **von Neumann bottleneck**. That is why caches exist.



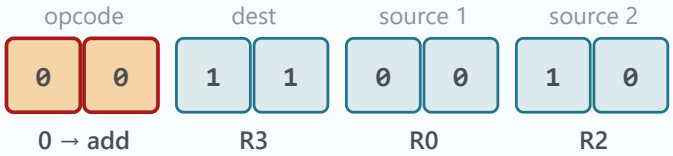
By contrast, the **Harvard architecture** keeps instruction and data memory separate; many microcontrollers, such as the AVR in an Arduino, use it.

Instructions Are Numbers Too: Encoding

Take a toy computer: **8 bits**, **4 registers** (R0–R3), **256** 8-bit memory cells and **4 instructions**: add (00), subtract (01), multiply (10), divide (11).

- The first 2 bits are the **opcode** (4 instructions → 2 bits).
- The next 2 bits are the **destination register**.
- The last 4 bits are two **source registers** (2 bits each).

"Add R0 and R2, store the result in R3" = **0 3 0 2** → `00110010`



The processor's "decode" step in Java: division by place value!

```
int instr = 0b00110010;
int opcode = instr / 64;      // first 2 bits
int dest = (instr / 16) % 4;   // next 2 bits
int src1 = (instr / 4) % 4;  // next 2 bits
int src2 = instr % 4;         // last 2 bits
System.out.println(instr);
System.out.println(opcode + " " + dest
    + " " + src1 + " " + src2);
```

Output:

```
50
0 3 0 2
```

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
2 int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;   // next 2 bits
4 int src1 = (instr / 4) % 4;  // next 2 bits
5 int src2 = instr % 4;         // last 2 bits
6 System.out.println(instr);
7 System.out.println(opcode + " " + dest
8     + " " + src1 + " " + src2);
```



instr = 50

32 + 16 + 2 = 50

The instruction is an 8-bit number: `00 11 00 10` = 50. Its bits split into four pairs.

Step 1 / 7

VARIABLES

`instr = 50`

OUTPUT

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
▶ int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;    // next 2 bits
4 int src1 = (instr / 4) % 4;   // next 2 bits
5 int src2 = instr % 4;         // last 2 bits
6 System.out.println(instr);
7 System.out.println(opcode + " " + dest
8     + " " + src1 + " " + src2);
```



opcode = 0 → add

`50 / 64 = 0`

Dividing by 64 = 2^6 drops the right 6 bits, leaving the first 2: `00` = 0, i.e. **add**.

Step 2 / 7

VARIABLES

`instr = 50` `opcode = 0`

OUTPUT

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
2 int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;   // next 2 bits
4 int src1 = (instr / 4) % 4;  // next 2 bits
5 int src2 = instr % 4;         // last 2 bits
6 System.out.println(instr);
7 System.out.println(opcode + " " + dest
8     + " " + src1 + " " + src2);
```



dest = 3 → R3

`50 / 16 = 3, 3 % 4 = 3`

Dividing by 16 drops the right 4 bits (`0011` = 3); `% 4` keeps the last 2: 3 → **R3**.

Step 3 / 7

VARIABLES

`instr = 50` `opcode = 0` `dest = 3`

OUTPUT

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
2 int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;   // next 2 bits
4 int src1 = (instr / 4) % 4;  // next 2 bits
5 int src2 = instr % 4;         // last 2 bits
6 System.out.println(instr);
7 System.out.println(opcode + " " + dest
8     + " " + src1 + " " + src2);
```



src1 = 0 → R0

`50 / 4 = 12, 12 % 4 = 0`

Dividing by 4 drops the right 2 bits (`001100` = 12); `12 % 4` = 0 → **R0**.

Step 4 / 7

VARIABLES

`instr = 50` `opcode = 0` `dest = 3` `src1 = 0`

OUTPUT

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
2 int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;   // next 2 bits
4 int src1 = (instr / 4) % 4;  // next 2 bits
▶ int src2 = instr % 4;         // last 2 bits
6 System.out.println(instr);
7 System.out.println(opcode + " " + dest
8     + " " + src1 + " " + src2);
```



src2 = 2 → R2

50 % 4 = 2

`% 4` gives the last 2 bits directly: `10` = 2 → **R2**.

Step 5 / 7

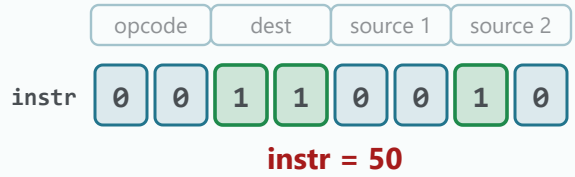
VARIABLES

`instr = 50` `opcode = 0` `dest = 3` `src1 = 0` `src2 = 2`

OUTPUT

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
2 int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;   // next 2 bits
4 int src1 = (instr / 4) % 4;  // next 2 bits
5 int src2 = instr % 4;         // last 2 bits
▶ System.out.println(instr);
7 System.out.println(opcode + " " + dest
8     + " " + src1 + " " + src2);
```



instr = 50

The instruction prints as a number: 50.

Step 6 / 7

VARIABLES

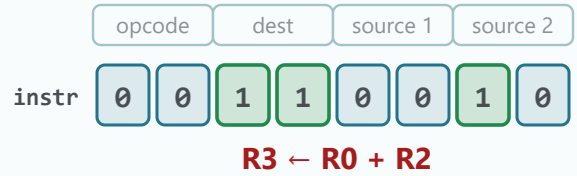
`instr = 50` `opcode = 0` `dest = 3` `src1 = 0` `src2 = 2`

OUTPUT

```
50
```

Decoding an Instruction: Code Trace

```
1 int instr = 0b00110010;
2 int opcode = instr / 64;      // first 2 bits
3 int dest = (instr / 16) % 4;   // next 2 bits
4 int src1 = (instr / 4) % 4;  // next 2 bits
5 int src2 = instr % 4;         // last 2 bits
6 System.out.println(instr);
▶ System.out.println(opcode + " " + dest
▶     + " " + src1 + " " + src2);
```



Fields: `0 3 0 2`. To the processor, the number 50 means “R3 ← R0 + R2”.

Step 7 / 7

VARIABLES

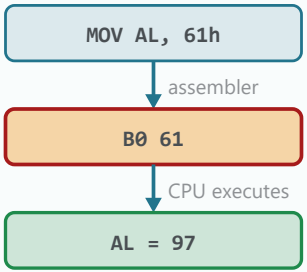
`instr = 50` `opcode = 0` `dest = 3` `src1 = 0` `src2 = 2`

OUTPUT

```
50
0 3 0 2
```

Assembly Language

- Assembly language is specific to a **processor architecture**: code written for x86 won't run on ARM.
- Each line usually corresponds to **one** machine instruction.
- It matters for understanding how the computer works at a low level.
- It is used for **performance tuning** and talking directly to hardware (drivers, embedded systems).



x86 instruction	Meaning	Machine code
`MOV AL, 61h`	move 61h (= 97) into register AL	`B0 61`
`MOV AX, BX`	copy the value in BX to AX	`66 89 D8`
`ADD EAX, 10`	add 10 to EAX	`83 C0 0A`
`XOR EAX, EAX`	clear EAX (x XOR x = 0)	`31 C0`

61h = 0x61 = 97 = ASCII `'a'`. Is a byte in memory a number, a letter or an instruction? That depends on **how the program uses it**.

Summary

Topic	Key points
Parts	CPU computes (ALU + control unit + registers); RAM is temporary, disk persistent; bus: address, data, control
Binary	bit = 0/1, 8 bits = 1 byte; d digits give 0 … $b^d - 1$; converting: divide by 2, read remainders bottom up
Negatives	two's complement: invert + 1; 8 bits: −128 … 127; past the limit, silent overflow
Floating point	fraction × $\text{base}^{\text{exponent}}$; IEEE 754 `float` / `double`; 0.1 + 0.2 ≠ 0.3
Text	ASCII 7 bits / 128 characters; Unicode code points (U+0000–U+10FFFF); UTF-8 1–4 bytes; Java `char` = 16-bit UTF-16
Structured info	image = RGB pixel array, sound = pressure samples, film = sequence of frames
Stored program	instructions are numbers in memory; PC = next address; fetch–decode–execute; jumps change the order
Levels	high-level language → assembly → machine code (bits); assembly is processor-specific, usually one instruction per line

SUMMARY

Common Mistakes

Literal out of range

`byte b = 200;` → `error: incompatible types: possible lossy conversion from int to byte`. A `byte` tops out at 127.

Forgetting the `L` suffix

`long x = 3000000000;` → `error: integer number too large`. Write `3000000000L`.

Mixing up char and String

`char c = "A";` → `error: incompatible types: String cannot be converted to char`. A char takes single quotes: `'A'`.

A leading zero

`int x = 010;` is octal, so its value is **8**! And `int y = 08;` won't compile (`error: ';' expected`): octal has no digit 8.

Ignoring overflow

`Integer.MAX_VALUE + 1` raises no error; it becomes **−2147483648**. Large products need `long`: `1920L * 1080`.

`==` on floating point

`0.1 + 0.2 == 0.3` → **false**. Check that the difference is small instead. Also, `"😀".length()` is 2, not 1.

SUMMARY

Sample Problems

1. Write 45 as an 8-bit binary number.
2. What is `10110110₂` unsigned? And signed, in 8-bit two's complement?
3. Write −20 in 8-bit two's complement.
4. What is `0x3F` in decimal, binary and octal?
5. Compute `01101101₂ + 00110110₂`. What is the result as a signed 8-bit `byte`?
6. What do `System.out.println('a' + 'b');` and `System.out.println("" + 'a' + 'b');` print?
7. What does the instruction `10011100` do on the toy computer?

▶ **Show answers**

SUMMARY

Exercises

1. What is the range of a 16-bit signed integer (Java `short`)?
2. Write the character codes of `"Java"` in decimal and binary.
3. How many bytes is an uncompressed 1024 × 768 image at 3 bytes per pixel?
4. What are `length()` and the UTF-8 byte count of `"ağaç"`?
5. Write a Java program that prints 0 to 15 as 4-digit binary numbers.
6. How many distinct bytes can a 32-line address bus address?

▶ **Answers / hints**

References

- D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, RISC-V Edition, 2nd ed., Morgan Kaufmann, 2020.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2nd ed., Microsoft Press, 2022.
- A. B. Downey and C. Mayfield, *Think Java*, 2nd ed., O'Reilly, 2019 (Chapter 1).
- Oracle, *The Java Tutorials*: “Primitive Data Types” and “Characters”.
- J. Gosling et al., *The Java Language Specification, Java SE 21*, §3.10 (Literals) and §4.2 (Primitive Types).
- The Unicode Consortium, *The Unicode Standard*, unicode.org; IEEE 754-2019, *Standard for Floating-Point Arithmetic*.
- J. von Neumann, *First Draft of a Report on the EDVAC*, 1945.

The End

Chapter 1.1: Computers

Next: Chapter 1.2: Computation and Computational Thinking