Programming Fundamentals 2

Pierre Talbot

23 February 2021

University of Luxembourg

Chapter II. Imperative Programming

A bottom-up approach

Types and Memory

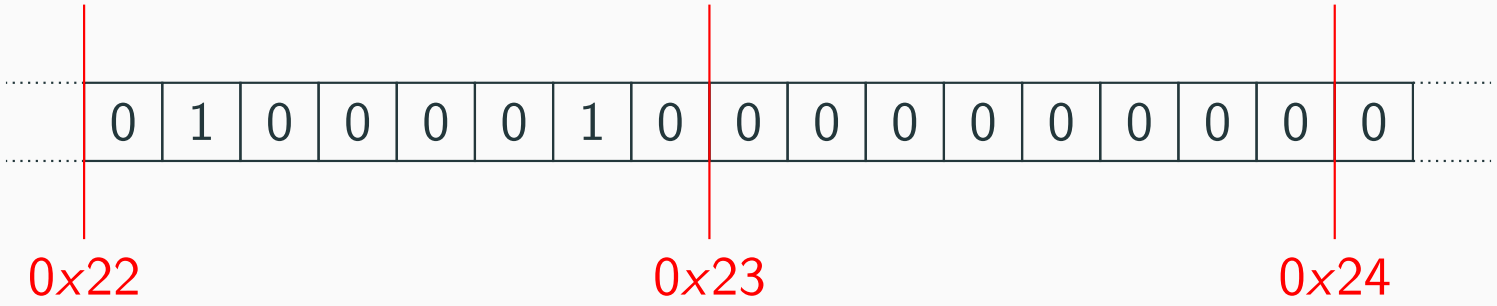
Untyped memory

The computer memory is just a big chunk of cells each containing either 0 or 1:

That is, the set $\{0,1\}^n$ where n is the size of your memory in bits. We say the memory is *untyped* since it contains only one sort of type ($\{0,1\}^n$).

Byte-addressable

Generally, the memory is divided into chunks of 8 bits, called bytes. Each byte has an address (usually written in hexadecimal form):



In a program, we read and write in the memory through variables and statements. But what is a variable really?

Mathematically speaking...

A *programming variable* can be seen as a predicate of the form $x \in T$ where x is its name and T is its type.

Type

A **type** is the set of values that a variable can take.

- `int` is the set $\{-2^{31}, \ldots, 0, \ldots, 2^{31} - 1\}$,
- `float` is the set $\{\ldots, -1.5, \ldots, -0, +0, \ldots, 1.125, \ldots, \texttt{NaN}\}$
  (precisely defined by the IEEE 740 standard),
- `char` is the set $\{\ldots, a, b, \ldots, \sum, \gamma, \ldots\}$,
  (precisely defined by the Unicode standard),
- `boolean` is the set $\{\mathit{true}, \mathit{false}\}$.

By `int x`, we mean $x \in$ `int`.

By `char c` we mean $c \in$ `char`.

Operationally speaking...

A *programming variable* is an address in memory (abstracted by a symbolic name) and a type.
A type is a size $s \in \mathbb{N}$ in bits and a pair of imaginary functions $f : \{0,1\}^s \to T$ and $g : T \to \{0,1\}^s$, such that T is the values you manipulate in the program.

Examples

- For `int`: size $= 32$ bits, $f_{int}(0^{24}01000001) = 64$,
- For `float`: size $= 32$ bits, $f_{float}(0^{24}01000001) = 9.108\ldots^{-44}$,
- For `char`: size $= 16$ bits, $f_{char}(0^{8}01000001) = A$,
- For `boolean`: size $= 1$ bit, $f_{boolean}(1) = \mathit{true}$.

More low-level details on memory representation and f in *Computing Infrastructure 1* (*e.g.* two-complement representation).

Static vs Dynamic Type

We say a programming language is *statically typed*, if each variable has a *single type* that can be figured out at compile-time. In contrast, it is *dynamically typed* if you can do something like `x = 4; x = "yo!";`—the type of `x` changes during the execution.

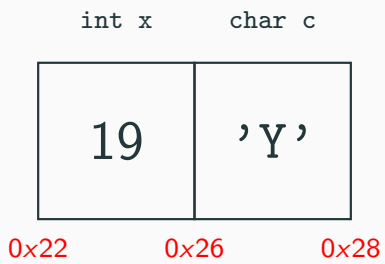
In Java, you must explicitly state the type of a variable when declaring it, and it cannot change later.

Drawing the memory

To simplify our drawings, we will view a cell in the memory as the content of a primitive variable (instead of a cell being just a bit).

```
int x = 19;
char c = 'Y';
```

will be represented as:



When not needed, we might not write the addresses and types explicitly.

Function and Evaluation Strategy

Previously...

```
import java.util.Scanner;

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("What's your name? ");
    String name = scanner.nextLine();
    System.out.print("What's your age? ");
    int age = scanner.nextInt();
    System.out.println("Welcome " + name + " (" + age
      + "years' old)");
    scanner.close();
  }
}
```

How to do if we want to get the information of a *second* person?

Copy-paste programming

You shouldn't do:

```
import java.util.Scanner;

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int age1 = ...
    String name1 = ...
    int age2 = ...
    String name2 = ...
    scanner.close();
  }
}
```

because you would have two times the same code!

(It is bad because if you fix a bug in the first part, you might forget to fix the copied/pasted second part.)

Using functions?

```
import java.util.Scanner;

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int age1, age2;
    String name1, name2;
    askPerson(scanner, name1, age1);
    askPerson(scanner, name2, age2);
    scanner.close();
  }

  static void askPerson(Scanner scanner, String name, int age) {
    System.out.print("What's your name? ");
    name = scanner.nextLine();
    System.out.print("What's your age? ");
    age = scanner.nextInt();
    System.out.println("Welcome " + name + " (" + age
      + "years' old)");
  }
}
```

Call-by-value evaluation strategy

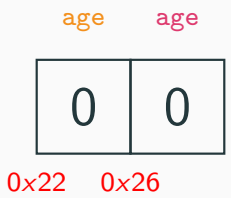
What happens when you pass an argument to a function?

```
public static void main(String[] args) {
  int age = 0;
  askAge(age);
  System.out.println("Age:  " + age);
}

static void askAge(int age) {
  age = 12;
}
```

Call-by-value evaluation strategy

What happens when you pass an argument to a function?

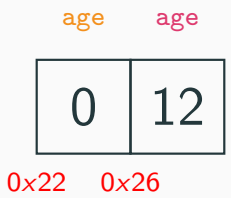
```
public static void main(String[] args) {
  int age = 0;
  askAge(age);
  System.out.println("Age:  " + age);
}

static void askAge(int age) {
  age = 12;
}
```



Call-by-value evaluation strategy

What happens when you pass an argument to a function?

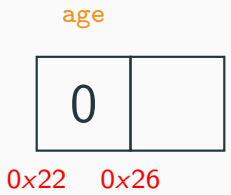
```
public static void main(String[] args) {
  int age = 0;
  askAge(age);
  System.out.println("Age:  " + age);
}

static void askAge(int age) {
  age = 12;
}
```



Call-by-value evaluation strategy

What happens when you pass an argument to a function?

```
public static void main(String[] args) {
  int age = 0;
  askAge(age);
  System.out.println("Age: " + age);
}

static void askAge(int age) {
  age = 12;
}
```



Call-by-value evaluation strategy

What happens when you pass an argument to a function?

The value is **copied** in a new cell (the parameter) when passed as an argument! This is called *call-by-value* evaluation strategy. The fact that both cells have the same symbolic name does not mean they are equal!

How to do then??

Call-by-value evaluation strategy

For a single value (like age) you can write:

```
public static void main(String[] args) {
  int age = askAge();
  System.out.println("Age: " + age);
}

static int askAge() {
  return 12;
}
```

However, for multiple values (*e.g.*, the age and name), we need to group the data in a common structure.

Tuple Type

Tuple

The simplest way to group values is with the *tuple type*.

In Python, you could implement `askPerson` with:

```
def askPerson():
  print("What is your age?")
  age = input()
  print("What is your name?")
  name = input()
  return (age, name)

(age, name) = askPerson()
print(name + ", next year you'll be " + (age + 1))
```

However, since the types are dynamic, the tuple has the type `string * string`, thus `age + 1` will fail at runtime.
In a statically typed language, such as OCaml, you create a tuple with:

```
let askPerson(): string * int = ("Albert", 12)
let person = askPerson()
let next_year_age = person.0 + 1
(* ^ Ooops compile-time error: we try to add Albert and 1... *)
```

Tuple

Mathematically speaking...

The tuple is exactly the Cartesian product $T_1 \times T_2$ between two (or more) types T_1 and T_2.

- $\texttt{int} \times \texttt{boolean} = \{(0, \texttt{true}), (0, \texttt{false}), (1, \texttt{true}), \ldots\}$,
- $(0, \texttt{true}) \in \texttt{int} \times \texttt{boolean}$,
- $(13, \texttt{false}) \in \texttt{int} \times \texttt{boolean}$,
- $(\text{"}\textit{Albert}\text{"}, 13) \in \texttt{String} \times \texttt{int}$

The field of a tuple is accessed with a projection `t.i` where $i \in \mathbb{N}$, *e.g.*, `person.0`, `person.1`, and $(0, \texttt{true}).1 = \texttt{true}$.

Oh BTW, in Java, there is no tuple type.

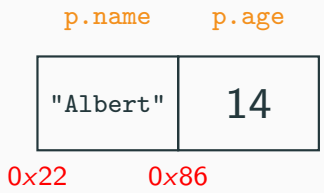
Record Type

Record

The record type is a simple extension to the tuple type which **explicitly names the fields of the tuple**. This is one of the most common constructions to group values in programming languages.

In C, you write:

```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  printf("Hello %s\n", p.name);
}
```

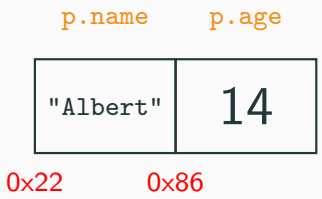


Mathematically, it remains a Cartesian product where the order of the components does not matter anymore.

Records as function parameters

What happens when you pass a record to a function?

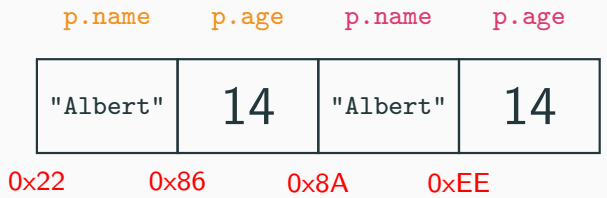
```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(p);
  printf("%s is %d years' old", p.name, p.age);
}

void birthday(Person p) {
  printf("Happy birthday %s", p.name);
  p.age = p.age + 1;
}
```



Records as function parameters

What happens when you pass a record to a function?

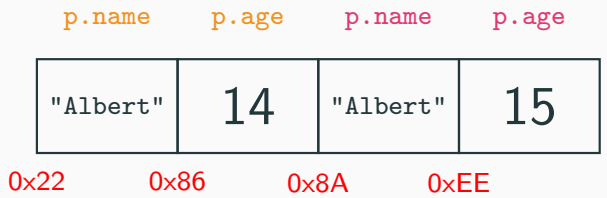
```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(p);
  printf("%s is %d years' old", p.name, p.age);
}

void birthday(Person p) {
  printf("Happy birthday %s", p.name);
  p.age = p.age + 1;
}
```



Records as function parameters

What happens when you pass a record to a function?

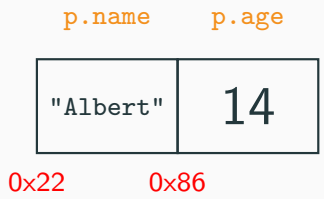
```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(p);
  printf("%s is %d years' old", p.name, p.age);
}

void birthday(Person p) {
  printf("Happy birthday %s\n", p.name);
  p.age = p.age + 1;
}
```



Records as function parameters

What happens when you pass a record to a function?

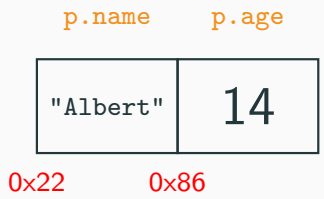
```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(p);
  printf("%s is %d years' old\n", p.name, p.age);
}

void birthday(Person p) {
  printf("Happy birthday %s\n", p.name);
  p.age = p.age + 1;
}
```



Records as function parameters

What happens when you pass a record to a function?

```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(p);
  printf("%s is %d years' old\n", p.name, p.age);
}

void birthday(Person p) {
  printf("Happy birthday %s\n", p.name);
  p.age = p.age + 1;
}
```



In C, a record is passed by value similarly to primitive types.

So how can we implement `birthday`?

Pointer Type

Pointer

We can copy the address of the value `p`, instead of copying the structure itself!

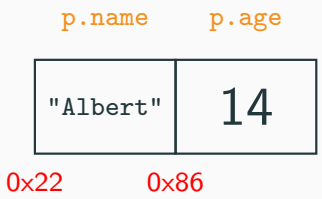
This is done through two important operators:

- The **address-of operator** `&x` returns the address of a variable x, *e.g.*, `&p` equals `0x22`.
- The **dereference operator** `*x` interprets the content of x as an address and returns the value at this address.
- Property: `*(&x) = x`.

Variables that contains addresses are called *pointer*.

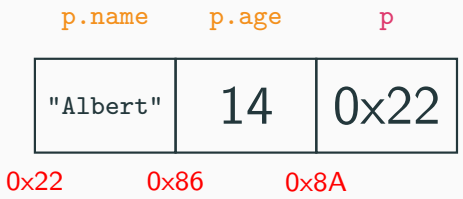
One nice trick: passing the address

```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(&p);
  printf("%s is %d years' old", p.name, p.age);
}

void birthday(Person* p) {
  printf("Happy birthday %s", (*p).name);
  (*p).age = (*p).age + 1;
}
```



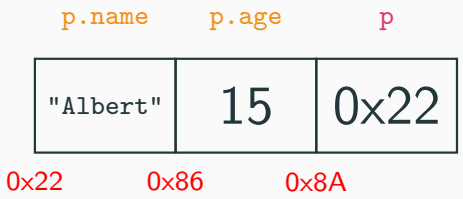
One nice trick: passing the address

```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(&p);
  printf("%s is %d years' old", p.name, p.age);
}

void birthday(Person* p) {
  printf("Happy birthday %s", (*p).name);
  (*p).age = (*p).age + 1;
}
```



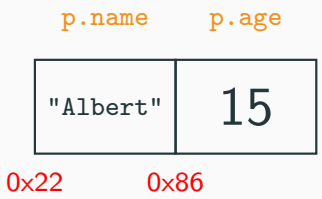
One nice trick: passing the address

```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(&p);
  printf("%s is %d years' old", p.name, p.age);
}

void birthday(Person* p) {
  printf("Happy birthday %s\n", (*p).name);
  (*p).age = (*p).age + 1;
}
```



One nice trick: passing the address

```
struct Person {
  char name[100];
  int age;
};

int main() {
  Person p  = {"Albert", 14};
  birthday(&p);
  printf("%s is %d years' old\n", p.name, p.age);
}

void birthday(Person* p) {
  printf("Happy birthday %s\n", (*p).name);
  (*p).age = (*p).age + 1;
}
```



Java does not have mutable record type or explicit pointer.

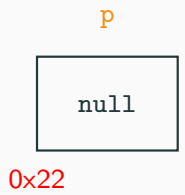
However, Java has:

- Implicit pointer called **reference**.
- An extension of the record type called **object**.
- **Immutable record** (new in Java 16, not covered here).

Reference Type

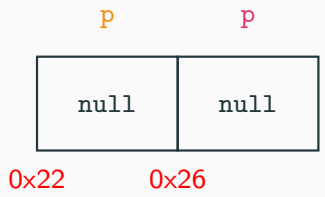
A first glimpse to objects (as records)

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name?  ");
    p.name = scanner.nextLine();
    System.out.print("What's your age?  ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



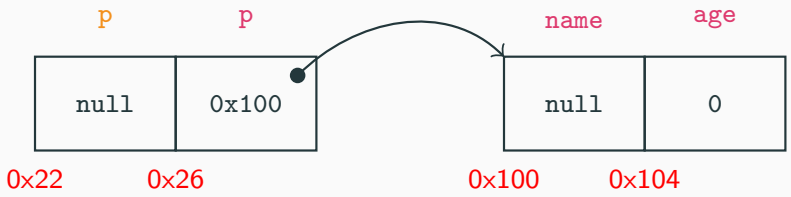
A first glimpse to objects (as records)

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name?  ");
    p.name = scanner.nextLine();
    System.out.print("What's your age?  ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



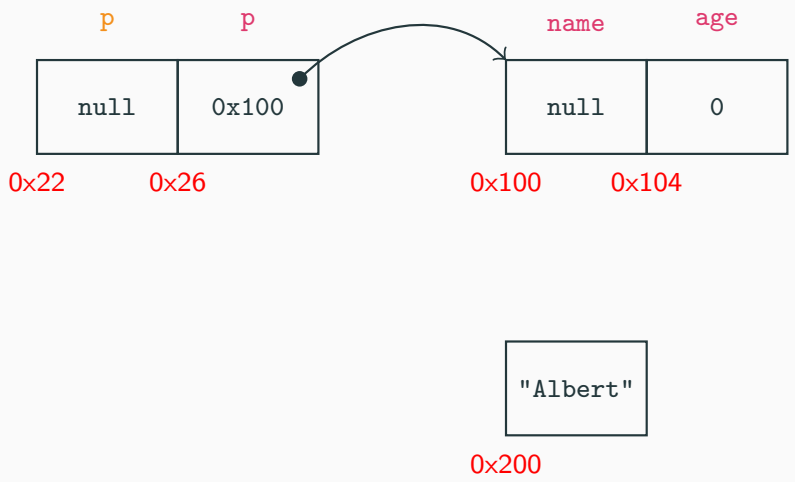
A first glimpse to objects (as records)

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name?  ");
    p.name = scanner.nextLine();
    System.out.print("What's your age?  ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



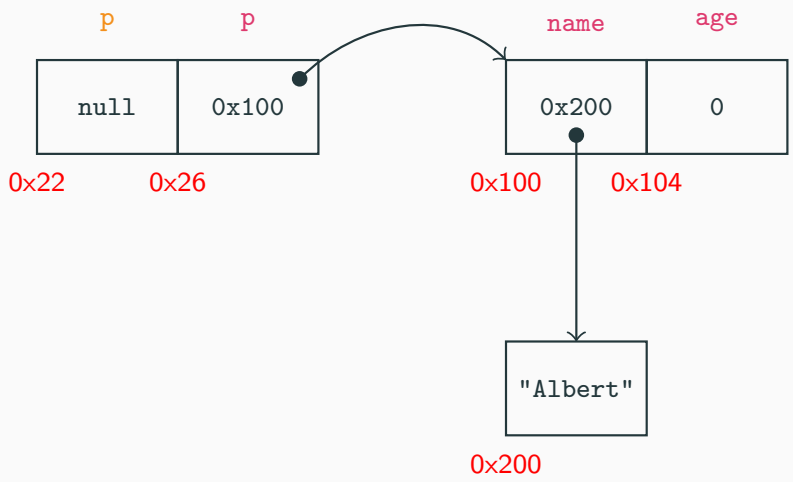
A first glimpse to objects (as records)

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name? ");
    p.name = scanner.nextLine();
    System.out.print("What's your age?  ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



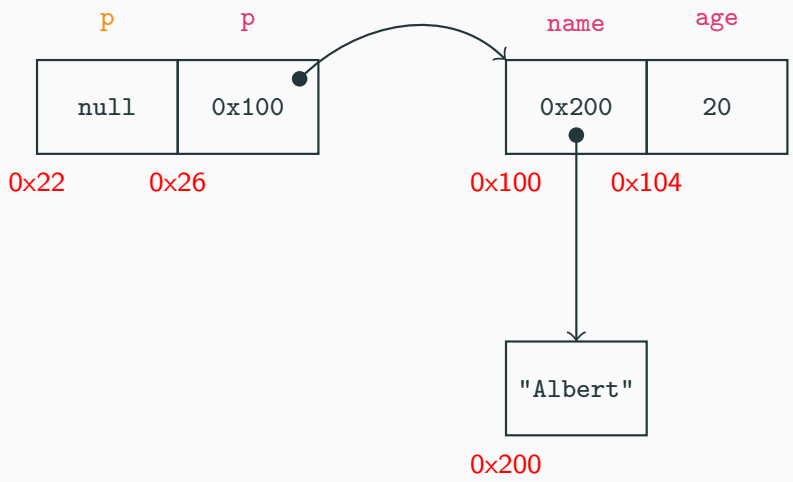
A first glimpse to objects (as records)

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name? ");
    p.name = scanner.nextLine();
    System.out.print("What's your age?  ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



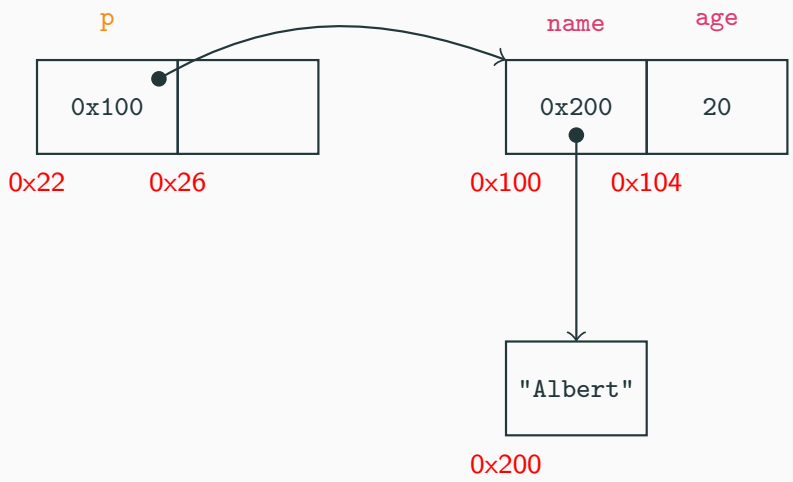
A first glimpse to objects (as records)

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name? ");
    p.name = scanner.nextLine();
    System.out.print("What's your age? ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



A first glimpse to objects (as records)

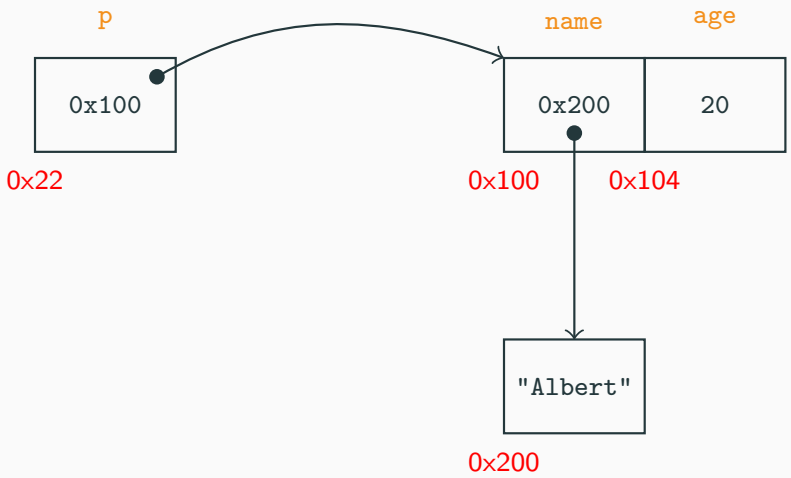
```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    Person p = new Person();
    System.out.print("What's your name? ");
    p.name = scanner.nextLine();
    System.out.print("What's your age? ");
    p.age = scanner.nextInt();
    scanner.nextLine();
    System.out.println("Welcome " + p.name + " (" + p.age
      + " years' old)");
    return p;
  }
}
```



Passing Object to Function

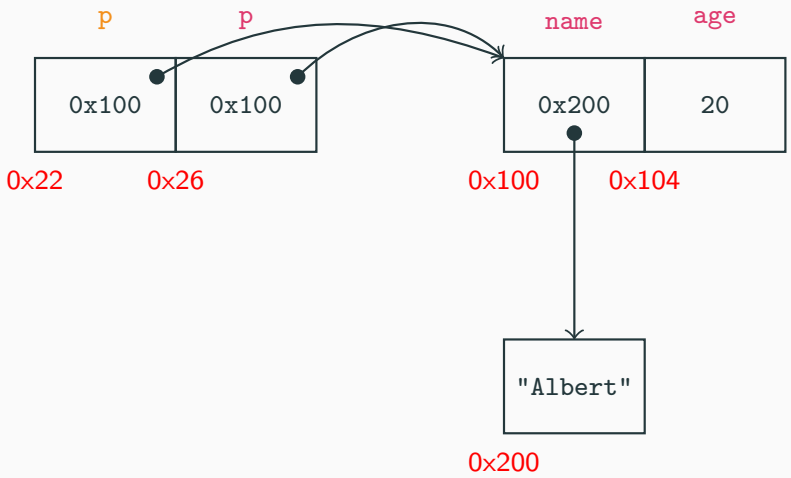
Passing reference by value

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    birthday(p);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    ...
  }

  static void birthday(Person p) {
    System.out.println("Happy birthday " + p.name);
    p.age = p.age + 1;
  }
}
```



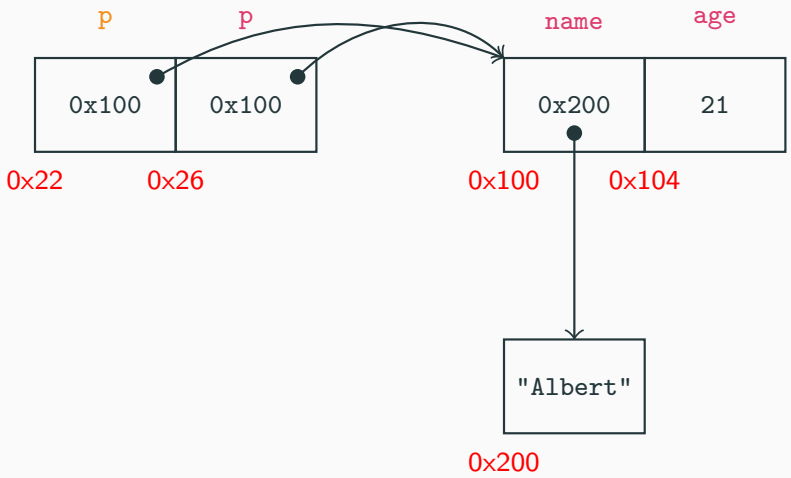
Passing reference by value

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    birthday(p);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    ...
  }

  static void birthday(Person p) {
    System.out.println("Happy birthday " + p.name);
    p.age = p.age + 1;
  }
}
```



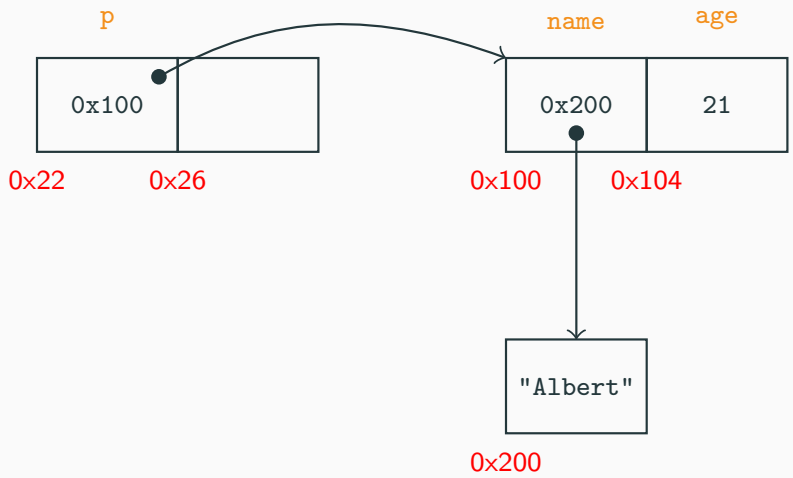
Passing reference by value

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    birthday(p);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    ...
  }

  static void birthday(Person p) {
    System.out.println("Happy birthday " + p.name);
    p.age = p.age + 1;
  }
}
```



Passing reference by value

```
class Person {
  public String name;
  public int age;
}

public class HelloWorld {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Person p = askPerson(scanner);
    birthday(p);
    scanner.close();
  }

  static Person askPerson(Scanner scanner) {
    ...
  }

  static void birthday(Person p) {
    System.out.println("Happy birthday " + p.name);
    p.age = p.age + 1;
  }
}
```



Summary on references

- The operator **new Person()**:
  1. Allocates a memory block and returns its address.
  2. Initializes the content by calling the constructor by default.
- `null` is the value put inside the memory cell of an uninitialized object, for instance: `Person p;`
- When passed by argument or returned, **only the address of the object is copied**, not its content.

In comparison to C...

- **Pointers are abstracted**: we do not need the operators `&x` or `*x`.
- Memory is allocated with `new`, but automatically freed by the *garbage collector*.

The Concert App

The Concert App

We write an app to manage the planning of concerts in *imperative Java*. This is how you would write such an app in a language such as `C`, thus **you should not imitate this style** using Java. Our goal is to compare the imperative/procedural style with the object-oriented style presented in the next chapter.

We use two records:

- A record `Concert`
- A record `ConcertPlanning`

```
// Invariant: startTime < endTime
public class Concert {
  public int startTime;
  public int endTime;

  public static Concert makeConcert(int startTime, int endTime) {
    assert startTime < endTime;
    Concert c = new Concert();
    c.startTime = startTime;
    c.endTime = endTime;
  }

  public static int duration(Concert concert) {
    return concert.endTime - concert.startTime;
  }
}
```

- *Defensive programming*: we add an assert in `makeConcert` to enforce the invariant.
- *Functions* are annotated with `static` and can be written inside the class, they are called *static methods*.

```
public class ConcertPlanning {
  public Concert[] concerts;

  public static ConcertPlanning makeConcertPlanning() { ... }
  public static void addConcert(Concert c) { ... }

  public static int totalTimeConcert(ConcertPlanning planning) {
    int total_time = 0;
    for(int i = 0; i < planning.concerts.length; ++i) {
      total_time += total_time(planning.concerts[i]);
    }
    return total_time;
  }
}
```

```
public class ConcertApp {
  public static void main(String[] args) {
    Concert c1 = Concert.makeConcert(18, 19);
    Concert c2 = Concert.makeConcert(20, 22);
    ConcertPlanning planning = ConcertPlanning.makeConcertPlanning();
    ConcertPlanning.addConcert(planning, c1);
    ConcertPlanning.addConcert(planning, c2);
    System.out.println("Total duration of the concerts: " +
      ConcertPlanning.totalTimeConcert(planning));
  }
}
```

We call static methods with the name of the class followed by the name of the function: `Class.method` (*e.g.*, `Concert.makeConcert`).