Programming Fundamentals 2

Pierre Talbot

27 May 2021

University of Luxembourg

Chapter 0. Getting Started

What is it?

This class will be about **programming** in Java.

Some aspects of this class are experimental.

- No distinction between lectures and labs.
- Intensive first half: **70% of your grade in 2 months.**
- Feedback on what you produce (quick grading, code review, ...).
- Standard and competitive tracks.

Don't hesitate to help us to improve this class!

Organization

FULL REMOTE: Every Tuesday and Thursday, 8:00 to 9:30.

There is no difference between lectures and labs!

Class layout:

1. **Chapters**: The core notions of Java are divided into 15 chapters.
2. **Live coding**: You watch me coding something.
3. **Code analysis**: We look at your projects and review them.
4. **Crafting**: Learn how to use your tools!

Ezhilmathi Krishnasamy (aka. Mathi) is the TA of this class, he will take a good look at your code and discuss it during code analysis session.

Planning

Two tracks: standard track and competitive track.

Standard track

- 16/02–15/04: **4 labs**, 1 every two weeks (40% of your grade).
- 15/04 (14:00–17:00): **Exam** (30% of your grade).
- 15/04–16/05 (labs 5 and 6): You will fight in the **A.I. Arena** (30% of your grade).
- Beware: coding exam plus **oral exam** for redoing students (100%).

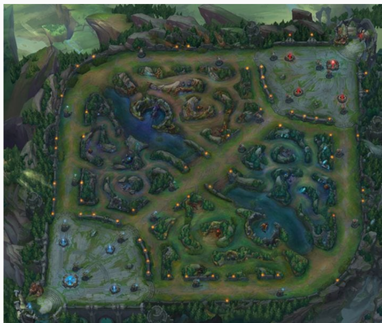
Labs

Three parts: basic exercises + main topic + competitive exercises.

- Lab 1: Connect Four
- Lab 2: Pokedeck
- Lab 3: Banking system
- Lab 4: Musical Improvisation

A.I. Arena

The remaining 30% will be gained by designing an artificial intelligence for a simplified version of a MOBA-like game.



You'll compete against each other for the throne!

Competitive track

- Track unlocked after you complete the standard track.
- Selected competitive exercices.
- You collect additional points.
- **Special events**: Hash code, Google Code Jam, ... (bonus points).

Competitive team

If you are interested, we can set up a team for ACM-ICPC in 1 or 2 years (need more or less preparation depending on your goal).

Competitive track

Coding competitions are very fun, and you learn a lot of new algorithms!

Competitive track planning

- 16th February → 15th April: Some **UVa problems** for each lab.
- **Team Event 1: Hash Code**: 25th February, 18:30
  `https://hashcodejudge.withgoogle.com`
- **Event 2: Google Code Jam Qualification**: 26th March, 23:00 to 28th March, 01:00
  `https://codingcompetitions.withgoogle.com/codejam`
- **Event 3: Google Code Jam Round 1A**: 10th April, 02:00–04:30
- **Event 4: Google Code Jam Round 1B**: 25th April, 17:00–19:30
- **Event 5: Google Code Jam Round 1C**: 1st May, 10:00–12:30
- **Event 6: Google Code Jam Round 2**: 15th May, 15:00–17:30

Those interested in the competitive track must **register here** (you can join anytime):

`https://docs.google.com/spreadsheets/d/`

`1KMZx58SoE08g-l4usphtaLFnPhKzBDhTpa9PgixOok8/edit?usp=sharing`.

What will you gain?

- Improve your programming craft, and code beautifully.
- Learn the basics of Java.
- Learn the basics of object-oriented programming.
- Feel more confident in the code you write.
- Develop your first 500-1K LOC programs.

Your coder toolbox

As a future professional software programmer, you need a decent (virtual) equipment! Here a list of what you need (for this class):

- **Shell**: Linux-compatible bash shell (aka. console or terminal)
- **Editor**: Sublime Text (`https://www.sublimetext.com/3`)
- **Java compiler/runtime**: `java` and `javac` commands
  Get *Open Java Development Kit* (Open JDK) (`https://www.oracle.com/java/technologies/javase-downloads.html`)
- **Source code control**: Git with `git` command.
  Also Github (`https://github.com`) as a collaboration platform built on top of git.
- **Build automation tool**: Maven with `mvn` command.
- **Communication**: Discord app.

No IDE for now. You must use Sublime text. IDEs are quite complicated and you don't know what's going on. We'll use one later.

Getting started

Depending on your system, the ways to install the tools are a bit different. Please, follow these videos according to your operating system (password: Programm1ng):

- Linux (Ubuntu):
  `https://unilu.webex.com/unilu/ldr.php?RCID=63896a9159d2a523118c1f724251cd0f`
- Mac OSX:
  `https://unilu.webex.com/unilu/ldr.php?RCID=8caf3ec8a59b40fd5721e74142c27e5c`
- Windows:
  `https://unilu.webex.com/unilu/ldr.php?RCID=5e77758fb1d61a90dca84802062d5fd0`

Try out as soon as possible Exercise 1 of Lab 1.
You **CANNOT** stay stuck at this stage.
Ask on Discord for any problem.

Getting help

Google to search for information (*e.g.*, *Java docs*, *Stackoverflow*, ...).

Discord (`https://discord.gg/SqarkmNQHe`) will be the privileged communication tool for questions.
Answer the questions of your peers, Mathi and I will answer too.
Here the different channels:

- `#tools`: for any installation trouble, *e.g.*, you can't run a Java program, and questions relevant to tooling.
- `#code`: all questions relevant to the code (from labs or classes).
- `#competition`: for the competitive track (UVa problems) and events (Google Hash Code, Code Jam).

By mail if your question is personal: `pierre.talbot@uni.lu`.

Do everything you can to find answers to your questions.

Resources

- **The Small Programming Handbook**: Cheat sheets on git, shell, Java pitfalls, Java conventions,... *Updated regularly* on `https://www.overleaf.com/read/tqxpqfwbbccc`
- Slides and recorded lectures, live coding and code analysis sessions.
- Tutorial on various topics inside the labs.

References

General Programming

- **Clean Code: A Handbook of Agile Software Craftsmanship**, Robert C. Martin
- Agile Software Development, Principles, Patterns, and Practices, Robert C. Martin
- Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
- The Mythical Man-Month: Essays on Software Engineering, Frederick Brooks

Java

- Effective Java 3rd Edition, Joshua Bloch
- Core Java Volume I - Fundamentals, Eleventh Edition, Cay S. Horstmann

Chapter I. Basics of Java Syntax

Syntax vs Semantics

- The **syntax** of a programming language defines the set of symbols allowed in the program, and its structure.
- The **semantics** of a programming language gives meaning to the sentences.

Examples

- Syntactically incorrect: "The eert is high" (unknown symbol "eert").
- Syntactically incorrect: "The tree is is high" (bad structure: repetition of "is").
- Syntactically correct but semantically incorrect: "The tree is reading a glass of water".

Syntax vs Semantics

It is similar with computer programs.

Examples

- Syntactically incorrect: `tni i = 1;` (unknown symbol "tni").
- Syntactically incorrect: `int i 1` (missing equility symbol).
- Syntactically correct but semantically incorrect: `int i = "a";` (expected type `int`, got `String`).
- Syntactically correct but semantically incorrect: `int i = 1; int i = 2;` (`i` redeclared).

The differences will be made precise in the class Programming Languages (BAINFOR-53).

Java Syntax

A language is a mix of various syntactic components such as:

- Statements
- Expressions
- Types
- Literals

When learning a language, we often look at examples, but this is not a formal nor complete specification of a language.
Therefore, we need a formalism to describe syntax: context-free grammar.

The syntax and informal semantics of Java is described in the *Java SE specification*:

https://docs.oracle.com/javase/specs/jls/se15/html/index.html

Statements (§14)

A **statement** is a construct that produces side-effect (e.g., it modifies the value of a variable, prints on the screen, ...).

```
Statement:
  Block
  LocalVariableDeclarationStatement  int x = 1;
                                     Integer o = new Integer(3);
  Assignment                         x = 3;
  IfThenStatement
  IfThenElseStatement
  WhileStatement
  ForStatement
  ClassDeclaration
Block:
  { [Statement] }                    int x = 1; x = x + 1;
IfThenStatement:
  if ( Expression ) Statement        if(x < 4) x = 2;
IfThenElseStatement:
  if ( Expression ) Statement else Statement
WhileStatement:
  while ( Expression ) Statement
ForStatement:
  for ( [ForInit] ; [Expression] ; [ForUpdate] ) Statement  for(int i = 0; i < n; i++) {
                                                                System.out.println(i);
                                                            }
ClassDeclaration:
  {ClassModifier} class TypeIdentifier [TypeParameters] [Superclass] [Superinterfaces] ClassBody
                                  public class Rectangle {
                                      private int width;
                                      private int height;
                                      ...
                                  }
```

Dangling `else` problem

Unlike Python, indentation is not mandatory in Java (that being said, you should indent as in Python).
This can lead to some problems with `if-else` statements (in C and C++ as well):

```
if (x > 0)
  if (y < 0)
    y = 2;
else
  x = 1;
```

The last `else` statement actually belong to the innermost `if`, here `if(y < 0)`.

To avoid ambiguity, always use curly braces:

```
if (x > 0) {
  if (y < 0) {
    y = 2;
  }
}
else {
  x = 1;
}
```

Expression (§15)

An **expression** is a code that evaluates to a value.

- Variable name: `i`, `x`, `average`.
- Array access: `arr[i]`, `matrix[i][j]`.
- Arithmetic expression: `7 + 8`, `x / 8 + 2 * 4`.
- Function call: `fibonacci(8)`.
- ...

Basically, if you can write `x = E;`, then `E` is an expression.

We will complete this list as we progress.

Types (§4)

Java is a **statically typed language**: a variable `x` has an *explicit* and *single* type during the execution of the program.

```
Type:
  PrimitiveType
  ReferenceType
PrimitiveType:
  (one of)
  boolean float double byte short int long char
ReferenceType:
  (see §4.3)       String, java.util.Scanner, ArrayList<Integer>
```

Arrays (§10)

1D array:

```
int n = 10;
int[] grades = new int[n]; // Create an array of size 'n', all elements are initialized to 0.
// ... Populate the array with grades (not shown)
int sum = 0;
// grades.length is an attribute of array giving the size of the array (here equals to 'n').
for(int i = 0; i < grades.length; ++i) {
  sum += grades[i];
}
System.out.println("The average of the student is "
  + (sum / grades.length));
```

2D array:

```
int n = 10;
int m = 29;
int[][] matrix = new int[n][m]; // Create a 2D array of size 'n * m', all elements are initialized to 0.
matrix[2][0] = 10; // Initialize the elements at coordinate (2,0) to 10.
```

Literals (§3.10)

Literals are the possible ground values in the language:

```
Literal:
  IntegerLiteral                2, 0, -1
  FloatingPointLiteral          1.1, 1.1f, 2., 2.9e-3
  BooleanLiteral                true, false
  CharacterLiteral              'a' '\u0370'
  StringLiteral                 "hello"
  TextBlock                     """ a very long multi-line string"""
  NullLiteral                   null
```

The set of literals is different according to the language, *e.g.*, in Python you have a literal for complex number (`3.14j`).

Unicode

Unicode is a standard to represent characters in a unified way.
`www.youtube.com/watch?v=-n2nlPHEMG8`

- Characters from more than 200 languages, but also emojis, are represented by a unique *code point*.
- For instance, a has the code point `U+0061`, and $\sum$ has `U+2211`.
- Code point can be encoded as:
  1. UTF-8: smaller string size, but linear array access (*e.g.*, no s[10]), because a character can occupies 1, 2, 3 or 4 bytes.
  2. UTF-16: 2 bytes per character, but constant array access.
- The Java `String` class uses UTF-16. You can write code points as `"\u1F602"`, which are automatically transformed into UTF-16.

Reading of the week: `https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/`

Floating-point numbers

The floating-point number `1.1` does not exist

But, sir, `System.out.println(1.1f)` is printing `1.1`! Well, kid, **it's a lie**! With enough precision `System.out.printf("%.10f", 1.1f);` will print `1.1000000238`.

Floating-point number are not exact!

Give you a treat, read this paper before you graduate:

What Every Computer Scientist Should Know About Floating-Point Arithmetic, David Goldberg, 1991

http://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/goldberg-floating-point.pdf

The (almost) smallest Java program

In order to execute some Java code, you absolutely need a `main` function.
It indicates where the program actually starts.

```
public class Chess {
  public static void main(String[] args) {
    System.out.println("Welcome to my Chess program");
  }
}
```

The file **must** have the same name as the class, here `Chess.java`.

Input/Output in 2 minutes

```java
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();
  }
}
```

We concatenate String with the operator +.

It works with literals and variables with primitive types as well.

Summary

We took a glimpse to some basic Java constructs.
You need nothing more to start coding your first Java programs :-)

Homework

- **Laboratory 1**, already available on Moodle.
- **Reading of the week**: `https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-ex`
- **Coding event** (optional)
  1. Register Google Hash Code (`hashcodejudge.withgoogle.com`).
  2. Give me the name of your team here: `https://docs.google.com/spreadsheets/d/1zSi6PG32kPGu5Kxhye19vsD7fqb2kqYVIntkh6Xd9fc/edit?usp=sharing`

Chapter II. Imperative Programming

A bottom-up approach

Types and Memory

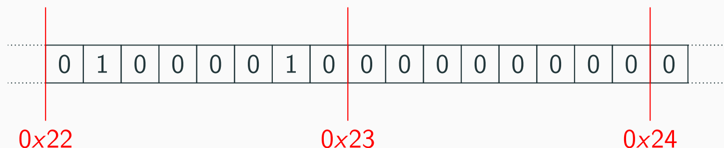
Untyped memory

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 $\{\ldots, -1.5, \ldots, -0, +0, \ldots, 1.125, \ldots, \mathtt{NaN}\}$
  (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 $\{true, false\}$.

By `int x`, we mean $x \in \mathtt{int}$.
By `char c` we mean $c \in \mathtt{char}$.

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, $f_{boolean}(1) = true$.

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.

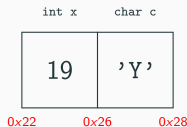
In Java, you must explicitly state the type of a variable when declaring it, and it cannot change later.

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

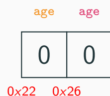
What happens when you pass an argument to a function?

```
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?

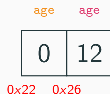
```
public static void main(String[] args) {
  int age = 0;
  askAge(age);
  System.out.println("Age:  " + age);
}

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



Call-by-value evaluation strategy

What happens when you pass an argument to a function?

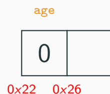
```
public static void main(String[] args) {
  int age = 0;
  askAge(age);
  System.out.println("Age:  " + age);
}

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.

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

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.

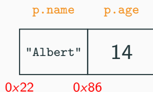
Record Type

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);
}
```

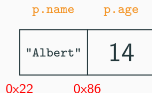


Mathematically, it remains a Cartesian product where the order of the components does not matter anymore.

Records as function parameters

What happens when you pass a record to a function?

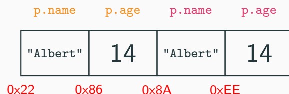
```
struct Person {
  char name[100];
  int age;
};

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?

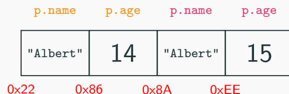
```
struct Person {
  char name[100];
  int age;
};

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?

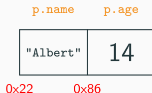
```
struct Person {
  char name[100];
  int age;
};

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?

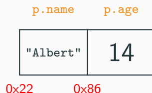
```
struct Person {
  char name[100];
  int age;
};

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!

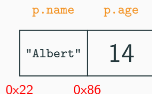
This is done through two important operators:

- 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*.

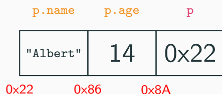
One nice trick: passing the address

```
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;
}
```



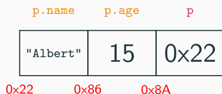
One nice trick: passing the address

```c
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;
}
```



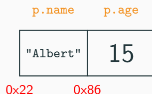
One nice trick: passing the address

```
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).

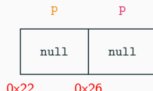
Reference Type

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;
  }
}
```

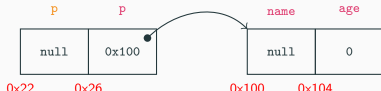
A first glimpse to objects (as records)

```
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;
  }
}
```



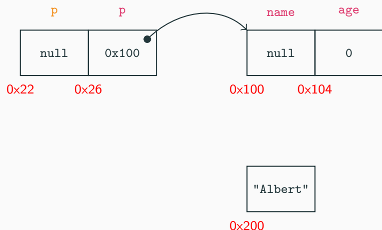
A first glimpse to objects (as records)

```
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;
  }
}
```



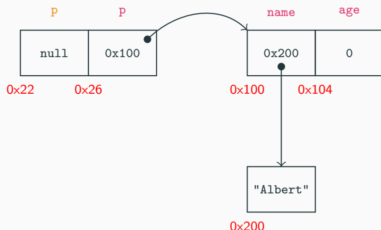
A first glimpse to objects (as records)

```
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;
  }
}
```



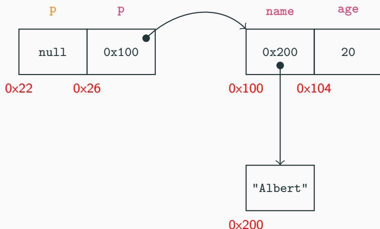
A first glimpse to objects (as records)

```java
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;
  }
}
```



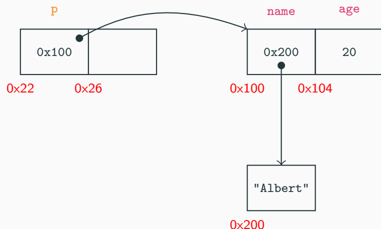
A first glimpse to objects (as records)

```java
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)

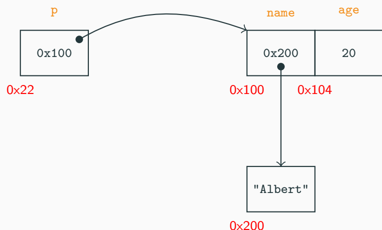
```java
class Person {
  public String name;
  public int age;
}

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

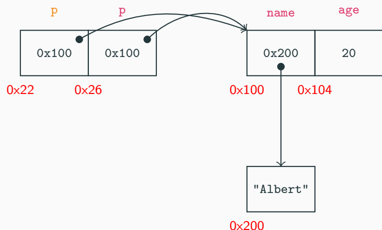
Passing reference by value

```
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;
  }
}
```



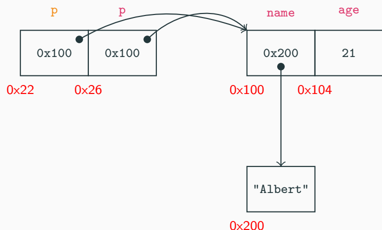
Passing reference by value

```
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;
  }
}
```



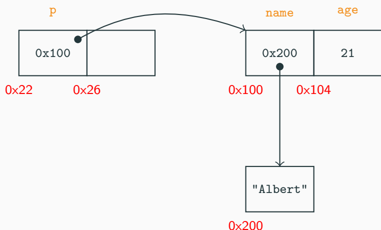
Passing reference by value

```
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

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`

```java
// 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*.

```java
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;
  }
}
```

```java
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`).

Chapter III. Object-Oriented Programming

Object type

Object = record + functions.

The functions are bundled together with the record, in a same structure.
In this context:

- the fields of the record are called *attributes*.
- the functions are called *methods*.

Characteristics of object-oriented programming

We can find 5 main characteristics of the object-oriented paradigm[1]:

- **Encapsulation**: attributes are not accessible from outside of the object.
- **Inheritance** (§4): An object I reuses the implementation of the methods of an object J, through class and *subclassing*.
- **Subtyping** (§6): If an object I contains, at least, all the attributes and method of J, then we can pass I as argument everywhere J is expected.
- **Multiple representations** (§8): a same set of methods (an *interface*) can be implemented by two different objects.
- **Open recursion**: A method on I can invokes any another method on the current object I.

[1] *Types and Programming Languages*, Benjamin C. Pierce.

Encapsulation

Concert app: from records to objects

Imagine that a newcomer in your team is asked to implement a function to cancel concerts:

```
public class ConcertPlanning {
  // ...

  // Start and end time of cancelled concerts are set to 0 and -1.
  public static void cancelConcert(ConcertPlanning planning, int concertIndex) {
    planning.concerts[concertIndex].startTime = 0;
    planning.concerts[concertIndex].endTime = -1;
  }
}
```

In this case, he decides that the time interval `[0..-1]` represents empty (cancelled) concert.

What could be the problem with this code?

The problem

The invariant `startTime < endTime` of the record `Concert` is violated. When computing `totalTimeConcert`, the total time will not be correct since the duration of each cancelled concert will be $-1 - 0 = -1$.

The problem is that anybody can access and modify the attributes of `Concert`. Imagine in large projects (hundreds of files): it is almost impossible to enforce the invariants everywhere, and bugs are easily introduced.

The solution: Encapsulation

A trait of object, namely *encapsulation*, helps in solving this problem. The previous code can be rewritten:

```
public class Concert {
  private int startTime;
  private int endTime;
  // ...
}
```

We set the attributes to `private`, we forbid any function external to this object to read or modify them. Any change to these attributes are located in the file `Concert.java`, so it is easier to enforce invariants.

```
public class ConcertPlanning {
  public static void cancelConcert(ConcertPlanning planning, int concertIndex) {
    // Won't work anymore! The assert will be triggered.
    planning.concerts[concertIndex] = Concert.makeConcert(0, -1);
  }
}
```

A paradigm shift

Due to encapsulation, there is a shift from data-centric programming to behavior-based programming:

- We do not reason on data anymore: these are hidden in classes that we cannot necessarily modify or even look at!
- We are only interested by **the services a class can give us**.
- These services are the methods of the class annotated with `public`.

Imperative / Procedural

```java
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;
  }
}
```

Object-oriented

```java
public class Concert {
  private int startTime;
  private int endTime;

  public Concert(int startTime, int endTime)
  {
    assert startTime < endTime;
    this.startTime = startTime;
    this.endTime = endTime;
  }

  public int duration() {
    return this.endTime - this.startTime;
  }
}
```

From records to objects

- `makeConcert` becomes the constructor `Concert` (**must have the same name as the class!**).
- Static methods become method (removes the `static` qualifier).
- The argument `Concert concert` passed to static methods, is now referred as **this** and is passed implicitly to the methods.

Using objects

Imperative / Procedural

```
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));
  }
}
```

Object-oriented

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

From records to objects

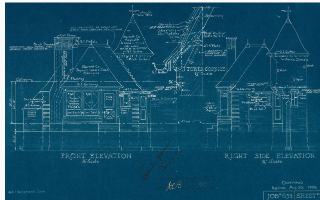
- The constructor is called with `new Concert(18, 19)`.
- `new Concert()` actually calls the *constructor by default*. It initializes the attributes to some default values (0 for integers, `null` for objects, ...).

Summary of objects

```
public class Concert {
  private int startTime;
  private int endTime;

  public Concert(int startTime, int endTime) {
    assert startTime < endTime;
    this.startTime = startTime;
    this.endTime = endTime;
  }

  public int duration() {
    return this.endTime - this.startTime;
  }
}

public class ConcertApp {
  public static void main(String[] args) {
    Concert c1 = new Concert(1, 2);
    Concert c2 = new Concert(3, 4);
    System.out.println("Duration c1: " + c1.duration());
    System.out.println("Duration c2: " + c2.duration());
  }
}
```

- Attributes of the class
- Constructor
- Methods of the class
- Constructor calls
- Method calls

Vocabulary: Class vs Instance vs Object



- A class is a "blueprint", a description of something, *e.g.*, the plan of a house.
- An object is the concrete realization of a class, *e.g.*, a house with a customized colour.
- An instance is a relationship between object and class, *e.g.*, the object `new Band("Daft Punk")` is an instance of the class `Band`.

A note on encapsulation

Encapsulation is not proper to the object-oriented paradigm. It exists in other paradigms, such as *functional programming* which often relies on modules to enforce encapsulation.
Moreover, some (non-mainstream) object-oriented languages do not enforce encapsulation (*e.g.*, CLOS).

Static methods and attributes

Static methods

Functions that do not compute on an object in particular.

```
public class Math {
  public static int max(int a, int b) {
    return a > b ? a : b;
  }
}
```

Consequently, in `static` methods, `this` is not passed as an argument to the function, thus you do not have access to the class attributes.

Advice

For now, you should stay away from static methods. They encourage an imperative/procedural style, instead of object-oriented style (see the examples above).

Static attributes

You can annotate an attribute with the keyword `static`. It creates a variable that is shared and global to all objects of this class.

We present two main use cases for static attributes:

1. Create unique identifiers.
2. Declare constant values.

Use case 1: unique identifier (UID)

Consider the following class:

```
public class Laptop {
  int product_uid;
  int ram;
  int hard_drive_memory;
  // ...
}
```

The attribute `product_uid` should be unique, each laptop should have a different one. How to ensure that?

Use case 1: unique identifier

Add a static attribute to count the number of objects created.

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  int hard_drive_memory;
  // ...
  public Laptop() {
    product_uid = uids;
    uids = uids + 1;
    //...
  }
}
```

Each time we will construct an object with `new Laptop()`, the static variable `uids` will be incremented, so the next invocation of the constructor will initialize `product_uid` to a different UID.

Use case 1: unique identifier

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  public Laptop(int ram) {
    this.product_uid = uids;
    this.ram = ram;
    uids = uids + 1;
  }
}

public class LaptopCreator {
  public static void main(String[] args) {
    Laptop l1 = new Laptop(500);
    Laptop l2 = new Laptop(1000);
  }
}
```

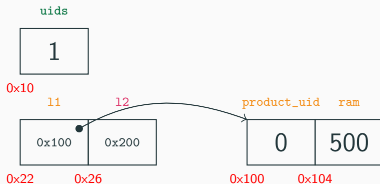
uids

0

0x10

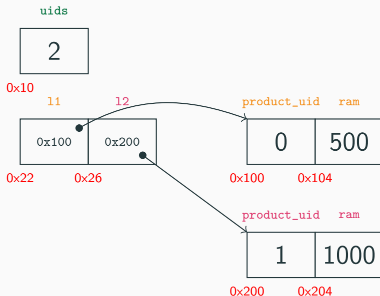
Use case 1: unique identifier

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  public Laptop(int ram) {
    this.product_uid = uids;
    this.ram = ram;
    uids = uids + 1;
  }
}

public class LaptopCreator {
  public static void main(String[] args) {
    Laptop l1 = new Laptop(500);
    Laptop l2 = new Laptop(1000);
  }
}
```



Use case 1: unique identifier

```
public class Laptop {
  static int uids = 0;
  int product_uid;
  int ram;
  public Laptop(int ram) {
    this.product_uid = uids;
    this.ram = ram;
    uids = uids + 1;
  }
}

public class LaptopCreator {
  public static void main(String[] args) {
    Laptop l1 = new Laptop(500);
    Laptop l2 = new Laptop(1000);
  }
}
```



Use case 2: constant values

```java
public class Checkers {
  static final int WHITE = 0;
  static final int BLACK = 1;
  int[][] grid;
  //...
  public printGrid() {
    for(int i = 0; i < grid.length; ++i) {
      for(int j = 0; j < grid[i].length; ++j) {
        if(grid[i][j] == WHITE) {
          System.out.print("\u25CB");
        }
        else {
          System.out.print("\u2B24");
        }
      }
      System.out.print("\n"); }}}
```

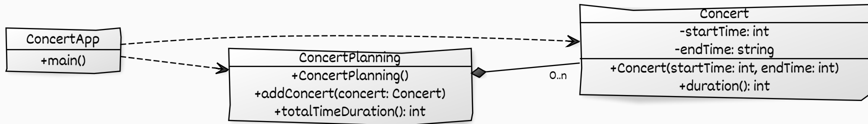
Final keyword

- `final int x` makes the content of the cell immutable (we cannot change it anymore).
- `final Person p` only means that we cannot assign *p* two times with `p = new Person()`. But we can still change what's inside of `p` with `p.name = "Alfred"`, as many times as we want (unless `name` is final as well).

Has-a relationships

Drawing class diagrams

When thinking about the design of an object-oriented application, it is often useful to draw a diagram showing how the classes are interconnected.



Unified Modeling Language (UML)

UML is a formalism to draw diagrams representing your application.

Class diagram

- A class is a box with three parts: the name, the attributes and the methods.
- *Visibility*: The symbols `-`, `+` or `#` in front of an attribute or method, respectively for `private`, `public`, `protected`.
- `public float m(int x)` is written `+m(x:int):float` in UML. The return type is written after the method.

Relationships between classes

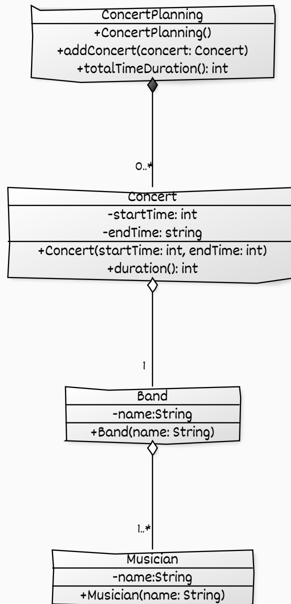
Relationships (simplified view)

- **Dependency** (dashed arrow): Class A uses class B in a method, but do not store object of type B as attributes.
- **Association** (plain line): Class A has an attribute of class B.
- **Aggregation** (arrow with empty diamond): Association + class B does not depend on A to live.
- **Composition** (arrow with plain diamond): Association + class B depends on A to live.

Each of these relationships can be annotated with *multiplicities*:

- `1`: Exactly one instance.
- `0..1`: Zero or one instance.
- `0..*`: Zero or more instances.
- `1..*`: One or more instances.

Aggregation vs Composition



More on UML relationships

Association, aggregation and composition all represent a connection between A and B such that B is an attribute of A.

- You don't need to (and probably shouldn't) specify all associations.
- You can use attributes instead of associations, if the link between the two classes is not important for what you want to show.
- An aggregation implies that an object is a part of another, *e.g.*, the band and the musician.
- An association would be, for instance, between the band and the musical style of the band.

UML should help you to explain the architecture of your project in a compact and informative manner.

Inheritance (is-a relationship)

The manga example

Suppose the following class:

```
public class Book {
  private String title;
  private Person writer;
  private int pages;
  public Book(String title, Person writer, int pages) {
    this.title = title;
    this.writer = writer;
    this.pages = pages;
  }
  public int numberOfPages() { return pages; }
  ...
}
```

Imagine we want to represent a *manga* (or comics). It is a book, but with at least one additional characteristic: the artist who drew it.

Let's pause a second, and think how we could do it with what we learnt.

Solution #1: Total rewrite

Perhaps the easiest solution is to create a totally new class `Manga`, not related to `Book`:

```
public class Manga {
  private String title;
  private Person writer;
  private Person artist;
  private int pages;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    this.title = title;
    this.writer = writer;
    this.artist = artist;
    this.pages = pages;
  }
  public int numberOfPages() { return pages; }
  ...
}
```

The problem is quite obvious: we have two classes (`Book` and `Manga`) with attributes and methods in common. *Don't repeat yourself (DRY)!*

Solution #2: Forwarding

A better solution is to store Book as an attribute of Manga:

```
public class Manga {
  private Book book;
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    this.book = new Book(title, writer, pages);
    this.artist = artist;
  }
  public int numberOfPages() { return book.numberOfPages(); }
  ...
}
```

We create *forwarding methods* in Manga to call the methods we need from Book. At least, if a bug is corrected in Book, we automatically benefit from the correction.

Solution #3: Inheritance

The concept of inheritance allows to "import" the attributes and methods from a *base class* `Book` into a *subclass* `Manga`.

```
public class Manga extends Book {
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    super(title, writer, pages);
    this.artist = artist;
  }
}

public class BookApp {
  public static void main(String[] args) {
    Manga snk = new Manga("Shingeki no kyogin", ..);
    System.out.println("Number of pages of SNK: "
      + snk.numberOfPages());
  }
}
```

Actually, it is not always simple to choose between solution #2 or #3 (cf. `https://en.wikipedia.org/wiki/Composition_over_inheritance`).

Inheritance subtlety #1: `super` keyword

The `super` keyword refers to the base class we inherited from. It can be used in three ways:

1. Call the constructor of the base class.
2. Refer to the public or protected attributes of the base class.
3. Refer to the public or protected methods of the base class.

```
public class Manga extends Book {
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    super(title, writer, pages);
    this.artist = artist;
  }
  public int blackAndWhitePages() {
    // We don't count the cover, two middle pages and last page.
    // (let's suppose it's really like that...)
    return super.numberOfPages() - 4;
  }
}
```

Inheritance subtlety #2: `protected` keyword

You can only use in the subclass `Manga` the attributes and methods with a `public` or `protected` visibility qualifier.

```
public class Manga extends Book {
  private Person artist;
  public Manga(String title, Person writer,
   Person artist, int pages)
  {
    super(title, writer, pages);
    this.artist = artist;
  }
  public void addOnePage() {
    // Will not work because 'pages' is private.
    super.pages = super.pages + 1;
  }
}
```

Just change `private` to `protected` if it makes sense:

```
public class Book {
  protected int pages;
  //...
}
```

Review and more

Preliminary

Here a example of inheritance we will reuse later:

```
class Weapon {
  protected double damage;
  public Weapon(double damage) {
    this.damage = damage;
  }
}

class Axe extends Weapon {
  private static final double DAMAGE = 10;
  public Axe() {
    super(DAMAGE);
  }
}

class Hammer extends Weapon {
  private static final double DAMAGE = 20;
  public Hammer() {
    super(DAMAGE);
  }
}
```

Quiz

- What are the attributes proper to an object and proper to a class?
- What is a constructor?
- What is `super` and `this`?
- Why does a constructor call must be the first instruction?
- What happens if we do not call `super` in the constructor of a subclass?

Solutions

Object vs Class

- Object attribute: `Weapon.damage`.
- An object attribute is different each time we create a new object, *e.g.*, `new Book()`.
- Class attributes: `Axe.DAMAGE` and `Hammer.DAMAGE`.
- A class attribute is shared by all objects, it is a kind of global variable.

Solutions

Constructor

- A constructor is not a method of the object. It is a function that construct the object.
- Therefore, constructors are not inherited (think: how could the constructor `Book` initialize the attribute `artist` in `Manga`?)

Solutions

`super` and `this` : 2 usages

- **During object construction**: call a constructor of the parent's class (`super(arg1, arg2)`) or of the current class (`this(arg1, arg2)`).
- The last case implies that a class can have more than 1 constructor (more on that in Chapter 4).
- **During method call**: `super` gives us a reference to the parent's object, and `this` to the current object, in order to explicitly access an attribute or method (`this.damage`).

Pitfall: shadowing

```
protected double damage;
public Weapon(double damage) {
  this.damage = damage;
}
```

Note: *Shadowing* of the attribute `damage` by the local variable, this is why we need to add `this.damage`.

Solutions

Call constructor of the base class first!

- We first build the base class, before building the subclass.
- Sometimes a bit cumbersome if you want to "prepare" the argument of the base class constructor.
  $\Rightarrow$ In that case, use a static method to prepare the arguments (see also *Builder pattern* in Chapter 1x).

Solutions

`super` not explicitly called

- The line `super();` is automatically inserted as the first line of the constructor body of the subclass.
- There is a compile-time error if there is no default constructor (constructor without arguments) in the base class.

Chapter IV. Ad-hoc Polymorphism

Polymorphism

- Fundamental concept in computer science.
- It means that "something can exist in different forms".
- A same type can have different behaviors.

Polymorphism

- Fundamental concept in computer science.
- It means that "something can exist in different forms".
- A same type can have different behaviors.

Different kind of polymorphisms

- Ad-hoc polymorphism.
- Subtyping polymorphisme (through inheritance).
- Casting polymorphisme.
- Parametric polymorphism (through generics).

Compile-time and runtime types

Compile-time type

Compile-time type

- The type is associated to the variable during the compilation.
- It is the type written when declaring a variable, *e.g.*, `Integer i`.
- Variables with primitive types can only have compile-time type.

Exercise: compile-time type

```
class WeaponStore{
  Weapon cheater = new Weapon(100);
  Weapon axe = new Axe();
  Weapon hammer = new Hammer();
  int number_weapons = 3;
  Number extra_damage = new Integer(42);

  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore();
store.price(new Axe());
store.price(new Weapon(22));
```

Solution: compile-time type

```
class WeaponStore{
  Weapon cheater = new Weapon(100); // Weapon
  Weapon axe = new Axe(); // Weapon
  Weapon hammer = new Hammer(); // Weapon
  int number_weapons = 3; // int
  Number extra_damage = new Integer(42); // Number

  // The compile-time type of w is Weapon
  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore(); // WeaponStore
store.price(new Axe()); // the temporary variable has type Axe.
store.price(new Weapon(22)); // temporary has type Weapon.
```

Runtime type

Runtime type

- The "real type" of the variable, as initialized at runtime.
- The runtime type (c_1) is always a subclass or identical ($c_1 \leq c_2$) to the compile-time type (c_2).
- For instance, `Axe axe = new Weapon(39);` does not make sense. A weapon *is not* an axe, a weapon can be many other things.
- Moreover, technically, how would we initialize the remaining members of `Axe`?

Example: runtime types

```java
class WeaponStore{
  Weapon cheater = new Weapon(100);
  Weapon axe = new Axe();
  Weapon hammer = new Hammer();
  int number_weapons = 3;
  Number extra_damage = new Integer(42);

  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore();
store.price(new Axe());
store.price(new Weapon(22));
```

Solution: runtime types

```
class WeaponStore{
  Weapon cheater = new Weapon(100); // Weapon
  Weapon axe = new Axe(); // Axe
  Weapon hammer = new Hammer(); // Hammer
  int number_weapons = 3; // int
  Number extra_damage = new Integer(42); // Integer

  // The dynamic type of w can be
  // Weapon, Axe or Hammer.
  public int price(Weapon w) { /* ... */ }
}
//... In main function.
WeaponStore store = new WeaponStore(); // WeaponStore
store.price(new Axe()); // the temporary variable has type Axe.
store.price(new Weapon(22)); // temporary has type Weapon.
```

Ad-hoc Polymorphism

Ad-hoc polymorphism (overloading)

Introductory challenge

- Create a class `Monster` and `Obstacle` each having a health points attribute and a method to decrease these health points.
- Add two methods to `Axe` and `Hammer` to attack the monsters and obstacles.
- The dammage of the axe on monsters is weigthed by 0.8, and on obstacles by 1.2.
- For the hammer, we have 1.4 and 0.7.

Thoughts on method names

Did you call the method to decrease the health points `set_life` or similarly?

Thoughts on method names

Did you call the method to decrease the health points `set_life` or similarly?

Coding style

- Methods such as `set_*` and `get_*` are *bad names* because they lead to imperative-style code, and not the "service-oriented" approach of OO.
- They somewhat break encapsulation because they expose internal attributes.
- A method should give a *service*, it must show in the name.
- It's hard to find good names, but very important.
- Sometimes, we want to have records (and not objects), in which case you can use immutable records or PODS and POJO, `http://en.wikipedia.org/wiki/Plain_old_data_structure`).

First solution

```
class Monster {
  private double life = 100;
  public void hit_me(double damage) { life = Math.max(0, life - damage); }
}
class Obstacle { /* similar */ }

class Axe extends Weapon {
  static final double MONSTER_DAMAGE_RATIO = 0.8;
  static final double OBSTACLE_DAMAGE_RATIO = 1.2;

  public void attack_monster(Monster m) {
    m.hit_me(damage * MONSTER_DAMAGE_RATIO);
  }

  public void attack_obstacle(Obstacle o) {
    o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
  }
  //...
}
class Hammer extends Weapon { /* similar */ }
```

Observation

```
public void attack_monster(Monster m)
```

Anything wrong with this method?

Observation

```
public void attack_monster(Monster m)
```

Anything wrong with this method?

Coding style

You should avoid any repetition, in the code, but also in the names.
This method signature already indicates we attack a monster, no need to repeat it.

Second solution

```
class Monster {
  private double life = 100;
  public void hit_me(double damage) { life = Math.max(0, life - damage); }
}
class Obstacle { /* similar */ }

class Axe extends Weapon {
  static final double MONSTER_DAMAGE_RATIO = 0.8;
  static final double OBSTACLE_DAMAGE_RATIO = 1.2;

  public void attack(Monster m) {
    m.hit_me(damage * MONSTER_DAMAGE_RATIO);
  }

  public void attack(Obstacle o) {
    o.hit_me(damage * OBSTACLE_DAMAGE_RATIO);
  }
  // ...
}
class Hammer extends Weapon { /* similar (constants change) */ }
```

Overloading

Definition

Overloading is a *compile-time mechanism* allowing us to use a same name for multiple methods, when those have a similar role.

Overloading

Definition

Overloading is a *compile-time mechanism* allowing us to use a same name for multiple methods, when those have a similar role.

Compile-time

It is only based on the compile-time type, the runtime type plays no role, and the method calls are resolved at compile-time (aka. *static binding*).

Overloading

When calling `obj.method(`a_1`, ..., `a_n`)`, how to be sure of which methods will be selected at compile-time? (trivial steps in grey).

1. Identify the classes to explore (compile-time type of `obj` + super classes).
2. Locate the *accessible* methods (public or protected in super classes) with the same name.
3. Select the methods with the same arity (numbers of arguments).
4. Select the *applicable* methods, *i.e.*, those with types of a_i are $\leq T_i$, T_i being the type of the parameter.
5. Apply an algorithm to select *the most specific method*.

Note: The return type does not matter.

Overloading resolution algorithm

This algorithm can be different depending on the language. Even between different versions of a same language (Java 1.2 vs Java 1.5 or later). Here, we present the most recent for Java.

1. Let A_i be the types of arguments, and P_i the types of the parameters.
2. For each argument, compute the "inheritance distance" between A_i and P_i, if $A_i \equiv P_i$ then the distance is 1.
3. Add distances.
4. The method with the smallest distance is selected.
5. If several distances are identical, then a compile-time error *ambiguous call* occurs.

Notes on overloading

- It is usually used when methods are non-ambiguous:
  - A different arity.
  - The parameters are not connected through inheritance.
- Otherwise, the programmer must manually execute the resolution algorithm to be sure of which method is called.
- Therefore, you should use it carefully and keep it simple.
- Generally, the philosophy adopted by the Java librairies.

Exercise I

Don't repeat yourself

Use a parent class `Destructible` extracting the common code in `Monster` and `Obstacle`.

Exercise I

Don't repeat yourself

Use a parent class `Destructible` extracting the common code in `Monster` and `Obstacle`.

Solution

```
class Destructible {
  protected double life = 100;
  public void hit_me(double damage) { life = Math.max(0, life - damage); }
}
class Monster extends Destructible { /* ... */}
class Obstacle extends Destructible { /* ... */ }
```

Exercise II

What is the method called, or the error, if for each object *o* declared below, we write `axe.attack(o)`?

```
class Axe {
  public void attack(Monster m) {} // (1)
  public void attack(Obstacle o) {} // (2)
  public void attack(Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();
```

Solution: Exercise II

```
Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(dmonster); // (3)
axe.attack(dobstacle); // (3)
axe.attack(monster); // (1)
axe.attack(obstacle); // (2)
```

Compile-time

Don't forget that *overloading* only looks at the compile-time type!

Exercise III

What about these examples?

```
class Axe {
  public void attack(Monster m, Obstacle o) {} // (1)
  public void attack(Destructible d, Monster m) {} // (2)
  public void attack(Monster m, Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(monster, obstacle);
axe.attack(dobstacle, monster);
axe.attack(dobstacle, dmonster);
axe.attack(dmonster, dmonster);
axe.attack(monster, monster);
axe.attack(monster, dobstacle);
```

Solution: Exercise III

```java
class Axe {
  public void attack(Monster m, Obstacle o) {} // (1)
  public void attack(Destructible d, Monster m) {} // (2)
  public void attack(Monster m, Destructible d) {} // (3)
}

Destructible dmonster = new Monster();
Destructible dobstacle = new Obstacle();
Monster monster = new Monster();
Obstacle obstacle = new Obstacle();

axe.attack(monster, obstacle); // (1)
axe.attack(dobstacle, monster); // (2)
axe.attack(dobstacle, dmonster); // error: no such method
axe.attack(dmonster, dmonster); // error: no such method
axe.attack(monster, monster); // error: ambiguous call between
                              // (2) and (3)
axe.attack(monster, dobstacle); // (3)
```

What to remember of ad-hoc polymorphism?

- Called polymorphism because a method can have several forms (all the methods with an identical name).
- *Overloading* mechanism allowing us to use a same name for different implementations. However, these methods should be connected semantically.
- The method called is chosen at compile-time (*static-binding*).

Chapter V. Subtype Polymorphism

Introductory challenge

Challenge

Add a method `ascii_art` returning the ASCII drawing of the weapon (`String` type).

Introductory challenge

Challenge

Add a method `ascii_art` returning the ASCII drawing of the weapon (`String` type).

```
class Axe { // ...
  // from http://www.chris.com/ascii/index.php?art=objects/axes
  public String ascii_art() {
    return
      " /'-./\_   \n" + // What's wrong here?
      ":    ||,>  \n" +
      " \.-'||    \n" + // And here?
      "     ||    \n" +
      "     ||    \n" +
      "     ||    \n";
  }
}
```

Introductory challenge (text block Java 15)

Challenge

Add a method `ascii_art` returning the ASCII drawing of the weapon (`String` type).

```
class Axe { // ...
  // from http://www.chris.com/ascii/index.php?art=objects/axes
  public String ascii_art() {
    return
      """
         /'-./\\_
        :    ||,>
         \\.-'||
             ||
             ||
             ||
      """;
  }
}
```

Subtype polymorphism

Shop

Consider a weapon shop `ArrayList<Weapon> store;`, can you print the ASCII drawing of all the weapons in this store?

Subtype polymorphism

Shop

Consider a weapon shop `ArrayList<Weapon> store;`, can you print the ASCII drawing of all the weapons in this store?

Issues

- Class `Weapon` doesn't have a method `ascii_art`!
- How to view the "real or concrete type" an object of type `Weapon`? More formally, how to view its runtime type (`Axe` or `Hammer`)? Spoiler: We don't! We use overriding instead so the runtime type is automatically used.

Overriding mechanism

Override-equivalent signatures

Two method signatures are *override-equivalent* if they have exactly the same name, same parameters types and return type. Actually, the return type can be covariant (we'll talk about that in Chapter 7).

Overriding mechanism

Override-equivalent signatures

Two method signatures are *override-equivalent* if they have exactly the same name, same parameters types and return type. Actually, the return type can be covariant (we'll talk about that in Chapter 7).

Overriding

For all classes $T \leq Weapon$, if a method $T.m$ is *override-equivalent* to $Weapon.m$, then the method called will be the one of the smallest subclass.

Overriding mechanism

Override-equivalent signatures

Two method signatures are *override-equivalent* if they have exactly the same name, same parameters types and return type. Actually, the return type can be covariant (we'll talk about that in Chapter 7).

Overriding

For all classes $T \leq Weapon$, if a method $T.m$ is *override-equivalent* to $Weapon.m$, then the method called will be the one of the smallest subclass.

Late-binding

Method calls are resolved at *runtime*. Indeed, we cannot guess at compile-time the runtime-type of the object. Why? Imagine the following code:

```
Weapon w;
if(a) { w = new Axe();} else { w = new Hammer(); }
w.ascii_art(); // Axe.ascii_art or Hammer.ascii_art?
```

Example overriding

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Design issue! A weapon cannot be draw in general. By the way, can a "general weapon" exist? Probably not since it is an abstract concept.

Example overriding

```
class Weapon {
  public String ascii_art() {
    return ????;
  }
}
```

Design issue! A weapon cannot be draw in general. By the way, can a "general weapon" exist? Probably not since it is an abstract concept.

Refactoring

- We must update the class `Weapon` to take into account the *new requirements*.
- Class `Weapon` must be an abstract class! An abstract class can contain attributes and methods, but some methods do not have a body.

Complete example

```
abstract class Weapon {
  protected double damage;
  public Weapon(double damage) {
    this.damage = damage;
  }
  abstract public String ascii_art();
}

class Axe extends Weapon {
  private static final double DAMAGE = 10;
  public Axe() {
    super(DAMAGE);
  }
  public String ascii_art() {
    return
      """
       <|>
        |
        |
      """;
  }
}
```

Complete example (next)

```
class Hammer extends Weapon {
  private static final double DAMAGE = 20;
  public Hammer() {
    super(DAMAGE);
  }
  public String ascii_art() {
    return
      """
       _ _
      |_|_|
        |
        |
      """;
  }
}

public class TestWeapon {
  public static void main(String[] args) {
    ArrayList<Weapon> store = new ArrayList<>();
    store.add(new Hammer());
    store.add(new Axe());
    for(Weapon w : store) {
      System.out.println(w.ascii_art());
    }
  }
}
```

What to remember about subtype polymorphism?

- "Polymorphism" because a type can have several forms (the subtypes, *i.e.*, in Java the subclasses).
- *Overriding mechanism* allowing to redefine a behavior more precisely.
- Methods are selected at runtime (*late-binding*).
- At compile-time, the methods are selected according to the rules of *ad-hoc polymorphism* and *overloading*.

Polymorphism Cocktail

Mixing overloading and overriding

- We can mix ad-hoc polymorphism and subtype polymorphism together.
- We first select the method via *overloading* (selected at compile-time).
- Then, at runtime, we check if *overriding* can apply (the signature must be *override-equivalent* to the one selected at compile-time).

Exercise

```
class A {
  void m(A x, B y){System.out.println ("1");}
  void m(B x, A y){System.out.println ("2");}
}
class B extends A {
  void m(B x, B y){System.out.println ("3");}
}
class C extends B {
  void m(B x, B y){System.out.println ("4");}
  void m(C x, C y){System.out.println ("5");}
  void m(B x, A y){System.out.println ("6");}
}
```

Exercise (part 2)

For each call, what is the method selected at compile-time, and then at runtime?

```
class PolymorphicCocktail {
  public static void main(String[] args) {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    a1.m(b1,c1);
    b1.m(b1,c1);
    c1.m(b1,c1);
    a1.m(a1,a1);

    a2.m(b1,c1);
    a3.m(b1,c1);
    b2.m(b1,c1);
    // ... (more in the next slide)
```

Exercise (part 3)

```
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;
    // ...

    a1.m(b2,a3);
    a2.m(b2,a3);
    a3.m(b2,a3);

    a1.m(c1,b1);
    b1.m(c1,b1);
    b2.m(c1,b1);
    c1.m(c1,b1);
  }
}
```

Correction

```
class PolymorphicCocktail {
  public static void main(String[] args) {
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    // solution of the form '(compile-time) / (execution-time)'
    a1.m(b1,c1); // ambiguous between (1) and (2)
    b1.m(b1,c1); // (3)/(3)
    c1.m(b1,c1); // (4)/(4)
    a1.m(a1,a1); // no suitable method found

    a2.m(b1,c1); // ambiguous between (1) and (2)
    a3.m(b1,c1); // ambiguous between (1) and (2)
    b2.m(b1,c1); // (3)/(4)

    a1.m(b2,a3); // (2)/(2)
    a2.m(b2,a3); // (2)/(2)
    a3.m(b2,a3); // (2)/(6)
    // ... (more in the next slide).
```

Correction (part 2)

```
    A a1 = new A();
    B b1 = new B();
    C c1 = new C();
    A a2 = b1;
    A a3 = c1;
    B b2 = c1;

    a1.m(c1,b1); // ambiguous between (1) and (2)
    b1.m(c1,b1); // (3)/(3)
    b2.m(c1,b1); // (3)/(4)
    c1.m(c1,b1); // (4)/(4)
  }
}
```

Complementary resources

The Java Language Specification

- Link: `http://docs.oracle.com/javase/specs/` (Java 15):
- §8.4.8: *overriding*.
- §8.4.9: *overloading*.
- §15.12: Method invocation (detailed steps performed by the compiler).
- Hard to read and understand because it is exhaustive!
- Nonetheless the best resource to find precise explanations.

Chapter VI. Casting Polymorphism

Casting of primitive types

Casting polymorphism

```
double price = 9.99;
int rounded_price = (int) price;
// rounded_price = ?
```

Casting

Casting is an operation allowing to convert a value from a type to a value of another type. For instance, to view `price` as an `int` instead of a `double`.

Recall from Chapter 2...

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, $f_{boolean}(1) = true$.

Bit-level Casting

We could just reinterpret the memory with the new type by changing the function f:

- Let `int x = 64;` and `float y = (float) x;`.
- We could view this operation as:
  $(\texttt{float})\texttt{x} = f_{float}(g_{int}(x)) = f_{float}(0^{24}01000001) = 9.108\ldots^{-44} = \texttt{y}$.

However, we would normally expect the casting operation to give $\texttt{y} = 64.0$ as a result.

Type-level Casting

- To reach the expected result, we introduce a casting function $cast : int \to float$.
- This function does not reinterpret the bits, but work at the level of the type T.
- Therefore, we have $cast(64) = 64.0$.
- There are cast functions for each conversion ($float \to int$, $char \to int$, ...).

Cast operations are partial functions

Some casting functions are partial functions (in theory):

- $cast : float \rightarrow int$: 4.5 can't be converted to integer.
- $cast : int \rightarrow short$: 100000 can't be converted to a short (too large).
- ...

In practice, they are some rules that make these functions total:

- $cast : float \rightarrow int$: round towards 0, *e.g.*:
  - $cast(4.5) = 4$
  - $cast(-4.5) = -4$
  - $cast(NaN) = 0$
- $cast : int \rightarrow short$: truncate the extra bits, and simply use f_{short} on the remaining bits:
  1. $g_{int}(100000) = 00000000\ 00000001\ 10000110\ 10100000$,
  2. $f_{short}(10000110\ 10100000) = -31072$

Implicit casting

To improve readability, many languages provide some automatic and implicit type conversions.

- Generally implicit when no precision is lost, *e.g.*, `short x = 10; int y = x`.
- Sometimes implicit although precision might be lost, *e.g.*, `int` to `float`.

Some languages such as Rust, forbids implicit casts, and favor explicit casts instead.

Casting of object types

Casting of object types

Following inheritance relationships, we can cast an object to a superclass or subclass.

- *Upcast* (implicit): Cast an object of type T to an object of type U such that $T \leq U$.

```
Weapon w = new Axe(); // The type Axe is upcasted to the type Weapon.
```

- *Downcast*: Cast an object of type T to an object of type U such that $T > U$.

```
Axe a = (Axe) w; // The type Weapon is downcasted from the type Weapon to the type Axe.
```

Downcast

Imagine the following code:

```
Weapon w = new Axe();
// ...
Hammer h = (Hammer) w; // oops!
```

- By downcasting, we cannot be sure that the runtime type of *w* is actually a type `Hammer`, in contrast to upcasting where the relationship can be verified at compile-time.
- In the previous example a `ClassCastException` is thrown.

Instanceof and getclass

When downcasting, you must always verify that the object you downcast is of the expected type. Suppose T is the runtime type of x:

- `x instanceof U` evaluates to *true* if $T \leq U$.
- `x.getClass() == U.class` evaluates to *true* if $T = U$.

Example (Instanceof vs getclass)

```
class MithrilAxe extends Axe { ... }
//...
Weapon w = new MithrilAxe();
if(w instanceof Axe) { System.out.println("w is an axe or a subtype of Axe.\n"); }
else if(w instanceof Hammer) { System.out.println("w is a hammer or a subtype of Hammer.\
// ...
if(w.getClass() == Axe.class) { System.out.println("w is an Axe."); }
else if(w.getClass() == MithrilAxe.class) { System.out.println("w is a MithrilAxe."); }
```

Is downcast a bad practice?

- Downcast is not necessarily a bad practice, however it leads to a more imperative programming style, and might indicate some issues with your object-oriented design.
- Nevertheless, downcast is always required for very specific cases such as overriding the method `equals`, see Chapter 7.

The expression problem

This simple discussion on downcast actually leads to a fundamental problem called *the expression problem*[2].

Extending data or operation?

- Casting polymorphism makes it easy to add new algorithms on existing data, without modifying existing code.
- Subtype polymorphism makes it easy to add new data classes without modifying existing algorithms.

It is best explained through an example: see *Live Coding Session: Coding a calculator!*
We will see in Chapter 10 the *visitor design pattern*, an object-oriented pattern that partially solves this problem.

[2] https://en.wikipedia.org/wiki/Expression_problem

What to remember about casting polymorphism?

- We can transform a value to view it under various forms.
- This form of polymorphism is probably the most widespread across languages (C, C++, Python, Javascript, ...).
- You must be careful to the specificities of each language. For instance in C++, there are 4 different casting operators (`static_cast` (type-level casting), `reinterpret_cast` (bit-level casting), ...).
- *Expression problem*: Tensions between data extension and algorithmic extension, and casting polymorphism vs subtype polymorphism.

Chapter VII. (Almost) Everything is Object

Object class

(Almost) Everything is object

All classes created automatically inherit from an existing class called `Object`.

```
class Weapon { ... }
// is equivalent to
class Weapon extends Object { ... }
```

This class contains many methods that can be overridden in the subclasses.

Almost?

Only primitive types are not objects, thus do not inherit from `Object`. But, they have object equivalent such as `Double`, `Integer`, ... For instance, the class `Integer`[3] is written as:

```
public class Integer {
  private int value;
  public Integer(int v) { this.value = v; }
  // ...
}
```

Autoboxing cast

Autoboxing is a special casting mechanism in Java to automatically cast a primitive type to its class equivalent:

```
static void f(Double d) { ... }
f(2.4); // call f with a primitive type automatically casted into Double.
```

The converse operation exists too and is called *unboxing*.

[3] https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/Integer.html

Object class

```
class Object {
  // Utility methods
  public String toString() { ... }
  protected Object clone() throws CloneNotSupportedException { ... }
  public boolean equals(Object obj) { ... }
  public int hashCode() { ... }
  // Thread related.
  public final void notify() { ... }
  public final void notifyAll() { ... }
  public final void wait() throws InterruptedException { ... }
  public final void wait(long timeout)
    throws InterruptedException { ... }
  public final void wait(long timeout, int nanos)
    throws InterruptedException { ... }
  // Garbage collector related.
  protected void finalize() throws Throwable { ... }
  // Reflection related.
  public final Class<?> getClass() { ... }
}
```

We will focus on the utility methods which are the most common.

Utility methods

The apparent simplicity of these methods is *deceptive* and they must be implemented carefully.

Effective Java, 3rd Edition, Joshua Bloch

- `toString`: Item 12: Always override `toString`
- `clone`: Item 13: Override `clone` judiciously
- `equals`: Item 10: Obey the general contract when overriding `equals`
- `hashCode`: Item 11: Always override `hashCode` when you override `equals`

Introductory exercise

Instructions

```
git clone https://github.com/ptal/PF2-lab-B.git
cd PF2-lab-B
mvn compile
mvn exec:java -Dexec.mainClass="lab.B.Main" -q
```

Add to the right classes (possibly `Weapon`, `Axe` and `Hammer`) the method `toString`:

```java
@Override public String toString() { ... }
```

Correction

toString

```
public abstract class Weapon {
  protected double damage;
  // ...
  @Override public String toString() {
    return damage + " damages";
  }
}
public class Hammer extends Weapon {
  // ...
  @Override public String toString() {
    return "Hammer of " + super.toString();
  }
}
```

- Although `Weapon` is only a concept, we implement `toString` in order to reuse its code in subclasses (DRY principle).
- To remember: always override `toString` (Item 12).

Copying

clone method

We distinguish three kinds of copies:

- Aliasing: Only the reference of the object is copied.

```
ArrayList<Integer> i = new ArrayList();
ArrayList<Integer> j = i;
j.add(3); // modify both i and j (see Chapters 2 and 3).
```

- Shallow copy: The attributes of the object are copied by aliasing.

```
public class IntegerList {
  public int x;
  public Array<Integer> y;
  public IntegerList shallowCopy() {
    return new IntegerList(x, y);
  }
}
// ...
IntegerList p1 = new IntegerList(3, new ArrayList());
IntegerList p2 = p1.shallowCopy();
p2.x = p2.x + 1; // modify only x in p2.
p2.y.add(3); // modify the array of both p1 and p2.
```

clone method

- Deep copy: The attributes are themselves copied using `clone`:

```
class IntegerList implements Cloneable {
  public int x;
  public Array<Integer> y;
  @Override public IntegerList clone() {
    return new IntegerList(x, y.clone());
  }
}
// ...
IntegerList p1 = new IntegerList(3, new ArrayList());
IntegerList p2 = p1.clone();
p2.x = p2.x + 1; // modify only x in p2.
p2.y.add(3); // modify the array of p2 only.
```

Additional comments on `clone`

- Usually, `clone` is implemented for deep copies. It is always good to mention the kind of copy in the documentation.
- The interface `Cloneable` must be used to indicate a class implements `clone`.
- According to Java documentation, although not strict requirements, you should have:

```
x.clone() != x
x.clone().getClass() == x.getClass()
x.clone().equals(x)
```

Your turn!

Add `clone` methods to the right classes in Lab B.

Covariant return type

When using `clone`, we must actually cast the object, *e.g.*,

```
Axe axe1 = new Axe();
Axe axe2 = (Axe)axe1.clone(); // Because clone returns an object.
```

We saw that two signatures are override-equivalent if they have the same name and the same parameters types. The return type can be *covariant*[4]: the return type can be a subtype of the return type of the parent's method. Therefore we can write:

```
@Override public Axe clone() // instead of Object clone()
```

while preserving override-equivalent signatures.

[4] http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29

Equality

What does it mean to be equal?

From a mathematical perspective, we define equivalent things in a set S using an equivalence relation $\theta \subseteq S \times S$.

Equivalence relation

- Reflexive: $\forall x \in S,\ (x, x) \in \theta$,
- Symmetric: $\forall x, y \in S,\ (x, y) \in \theta \Leftrightarrow (y, x) \in \theta$,
- Transitive: $\forall x, y, z \in S,\ (x, y) \in \theta \wedge (y, z) \in \theta \Rightarrow (x, z) \in \theta$.

- The method `boolean equals(Object other)` should specify an equivalence relation (not always possible however).
- By default, `Object.equals` only compares the references. This corresponds to the *smallest equivalence relation*.

Additional requirements

We have two additional requirements for equivalence relations over objects:

- `x.equals(null)` must always be `false`,
- Consistent: `x.equals(y) == x.equals(y)` (multiple invocations return the same result).

Why not `boolean equals(Axe a)` instead of `boolean equals(Object o)`

Because `equals` would not be override-equivalent to `Object.equals` anymore, which means that the implementation of `equals` is selected at compile-time by overloading.

Your turn!

- Implement `equals` for the right classes.
- Add a new class:

```java
public class MithrillAxe extends Axe {
  private boolean madeByDwarf;
  // ...
}
```

  Add the methods `toString`, `clone` and `equals` to this class, and test them.
- Can we guarantee `MithrillAxe.equals` to be an equivalence relation?

Implementing the `equals` method (part 1a)

```
public abstract class Weapon {
  @Override public boolean equals(Object o) {
    if(o == this) {
      return true;
    }
    else if (!(o instanceof Weapon)) {
      return false;
    }
    else {
      return ((Weapon)o).damage == damage;
    }
  }
}
```

Implementing the `equals` method (part 1b)

We must override `equals` in `Hammer` and `Axe` too, otherwise an axe and a hammer with the same amount of damages would be considered equal:

```
public class Axe extends Weapon {
  @Override public boolean equals(Object o) {
    return (o instanceof Axe) && super.equals(o);
  }
}
public class Hammer extends Weapon {
  @Override public boolean equals(Object o) {
    return (o instanceof Hammer) && super.equals(o);
  }
}
```

Implementing the equals method (part 2)

```
public class MithrillAxe extends Axe implements Cloneable {
  @Override public boolean equals(Object o) {
    if(o == this) {
      return true;
    }
    else if (!(o instanceof MithrillAxe)) {
      return false;
    }
    else {
      return super.equals(o)
        && ((MithrillAxe) o).madeByDwarf == madeByDwarf;
    }
  }
```

Unfortunately, this method is not symmetric:

```
MithrillAxe a1 = new MithrillAxe(true);
Axe a2 = new Axe();
a2.equals(a1); // is true, but
a1.equals(a2); // is false.
```

Implementing the `equals` method (part 3)

```
public class MithrillAxe extends Axe implements Cloneable {
  @Override public boolean equals(Object o) {
    if(o == this) {
      return true;
    }
    else if(o.getClass() == Axe.class) {
      return super.equals(o);
    }
    else if (!(o instanceof MithrillAxe)) {
      return false;
    }
    else {
      return super.equals(o)
        && ((MithrillAxe) o).madeByDwarf == madeByDwarf;
    }
  }
```

That's the best we can do, but this method is still not transitive:

```
MithrillAxe a1 = new MithrillAxe(true);
Axe a2 = new Axe();
MithrillAxe a3 = new MithrillAxe(false);
a1.equals(a2); // is true and
a2.equals(a3); // is true, but
a1.equals(a3); // is false.
```

A use-case of the `equals` method

The operation `ArrayList.contains` relies on `equals` to detect objects that are equal.

```
ArrayList<Weapon> store = new ArrayList<>();
store.add(new Axe());
store.add(new MithrillAxe(true));
System.out.println(store.contains(new Axe())); // prints true.
```

If `equals` is not implemented, the semantics of `contains` changes, and rely on reference equality.

Hash

Hashing

A hash is a function $hashCode : T \rightarrow \mathbb{N}$. That is, it maps a type `T` to an integer.

- `x.equals(y)` implies `x.hashCode() == y.hashCode()`.
- However, the converse is not necessarily true.
- To guarantee this implication, **you should always override `hashCode` if you override `equals`.**
- Normally, it is faster to compute a hash than comparing two objects.

An example

```java
public abstract class Weapon {
  protected int damage;
  @Override public int hashCode() {
    return damage;
  }
}

public class MithrillAxe extends Axe implements Cloneable {
  private boolean madeByDwarf;
  @Override public int hashCode() {
    return super.hashCode() * 10 + (madeByDwarf ? 1 : 0);
  }
}
```

In that case, we have `x.hashCode() == y.hashCode()` $\Leftrightarrow$ `x.equals(y)` (assuming no overflow). But that is not always the case, imagine the hash function of an array.

A use-case of `hashCode`: HashSet

`java.util.Set` is an interface for collection implementing the semantics of a (mathematical) set. It has the property that if `x.equals(y)`, then only x or y is in the set.

From a practical standpoint, a set can be implemented by various data structures with different tradeoff.

```
HashSet<Weapon> store = new HashSet<>();
store.add(new Axe());
store.add(new MithrillAxe(true));
store.add(new MithrillAxe(false));
store.add(new Axe());
System.out.println(store.size()); // prints 3.
```

Another possible choice is `TreeSet`, however this one needs the class to implement the `Comparable` interface (imposes a total order on the elements).

Summary

- Object is the parent class of all classes.
- Four essential methods to override from Object.
- A lot of subtleties... Necessary to study these to become proficient in Java.

Correction of the exercises: `git checkout correction` in your repository (you need to commit your changes first).

Chapter VIII. Error Management

Error management

We are going to study two aspects of error management:

- How to report and handle errors generated by a method.
- How to avoid unwanted errors by testing.

Error handling

Case study: search a Pokemon card

We are considering the laboratory 2 as an example. In this laboratory, we have a deck of cards and we want to search for a card meeting a criterion such as finding a card with a specific name.

An example of signature for this method is:

```
public Card searchByName(String name) { ... }
```

What to do if the card is not in the collection?

We look at various solutions, proposed in your labs, to this problem:

(I) Return a special value (*e.g.*, `null` or `-1`),

(II) Return the string representation of the card directly,

(III) Return an error object of the same type,

(IV) Return a list,

(V) Throw an exception.

Case Study Ia: Return `null`

```java
public Card getCardById(String id) {
  for (Card card : deck) {
    if(id.equals(card.getId())) {
      return card;
    }
  }
  return null;
}
```

If the card is not in the deck, `null` is returned.

Case Study Ib: Return -1

```java
public int findCardIndexByNumber(int cardNumber)
{
  int index = -1;
  for (int i=0; i < deck.size() ; ++i) {
    if (deck.get(i).getCardNumber() == cardNumber){
      index = i;
      break;
    }
  }
  return index;
}
```

Instead of directly returning a card, we return its index in the deck, and -1 if it is not inside.

Case Study I: Discussion

This kind of error handling is very common in language such as C. One problem with such scheme is on the calling site:

```
public void modifyCard(String id) {
  Card card = getCardById(id);
  if (card != null)
    card.modify();
}
```

- Each time the method is called, we must check for `card != null` or `cardIdx != -1`.
- We can easily forget to check that, and generate a `NullPointerException` or `OutOfBoundException`.
- Normally, **this solution is not the way to go**.
- See also *Code Clean*, Chapter 7, “Don’t return `null`”.

Case Study I: Discussion

Further, it leads to code harder to read when all the error handling is performed that way:

```
int id = askUserForID();
if(id == -1) {
  wrongUserInput();
}
else {
  Card card = getCardById(id);
  if(card == null) {
    wrongID();
  }
  else {
    // ...
  }
}
```

The logic of the code is lost in error handling, the code is actually quite simple:

```
int id = askUserForID();
Card card = getCardById(id);
// ...
```

Case Study II: Return the String representation

```java
public String searchCardByNumber(){
  System.out.println("What is the number you want to search?");
  String searchedNumber= input.getInput();
  for(int i=0;i < cards.size(); i++){
    if(cards.get(i).number.equals(searchedNumber))
      return "Searched card: \n" +cards.get(i).toString();
  }
  return "Card number doesn’t exist.";
}
```

Case Study II: Discussion

This solution has massive downsides:

- We cannot reuse `searchCardByNumber` for something else (e.g., modifying the card),
- We cannot use `searchCardByNumber` if we already obtained the ID from another source,
- The user interface is tightly coupled with the business logic: hard to maintain.
- This function has too many responsibilities: (i) ask a number to the user, (ii) search the number, (iii) prepare the resulting output.
- **This is not a good solution neither**.
- Bad variant: `searchCardByNumber` directly prints the message and returns nothing.

Case Study III: Return an error object of the same type

```
public Card searchByName(String name) {
  Card matching = new Card (" Not Found", " ", 0);
  for(int i=0; i < cards.size(); i++)
  {
    if (cards.get(i).getName().equals(name))
    {
      matching = cards.get(i);
      return matching;
    }
  }
  return matching;
}
```

Case Study III: Discussion

This solution is not too bad, but has a strong disadvantage:

This method expects to be called in a specific context.

That is, it expects the card to be printed immediately afterwards.

- What if we use this method in another context, *e.g.*, to find a card to modify?
- Should we let the user modify a card that doesn't exist?
- How to detect the card doesn't exist?

Note that in some other places, an improvement of this solution can be good, c.f., Code Clean, Chapter 7, "Define the Normal Flow".
See also the SPECIAL CASE DESIGN PATTERN.

Case Study IV: Return a list

```
public ArrayList<Card> getCardsByName(String name) {
  ArrayList<Card> searchResults = new ArrayList<Card>();
  for (Card card : deck) {
    if(name.equals(card.getName())) {
      searchResults.add(card);
    }
  }
  return searchResults;
}
```

Case Study IV: Discussion

This solution is a good one:

- It returns an empty list if no card matches the name,
- It is callable in any kind of context,
- It generalizes the previous method to cards with multiple names (why can it happen though?).

See also *"Item 54: Return empty collections or arrays, not nulls"*, *Effective Java.*

Case Study IVb: Return a Optional

```
public Optional<Card> getCardByName(String name) {
  for (Card card : deck) {
    if(name.equals(card.getName())) {
      return Optional.of(card);
    }
  }
  return Optional.empty();
}
```

Case Study IVb: Discussion

On the calling site, we cannot forget to check that the card exists (unlike with `null` and `-1`), because this information is built in the return type.

```
Optional<Card> card_opt = getCardByName(name);
if(card_opt.isPresent()) {
  Card card = card_opt.get();
}
else {
  // ...
}
```

However, the code can become less clear (similarly than with return code). The methods `Optional.ifPresent`, `Optional.map`,... can help for this purpose.

See also *"Item 55: Return optionals judiciously", Effective Java.*

Case Study V: Throw an exception

```
public Card searchCardByName(String name){
  for (Card card : cards) {
    if(card.name().equal(name)) {
      return card;
    }
  }
  throw new RuntimeException("Card not found");
}
```

Case Study V: Discussion

This is the most idiomatic way of reporting an error in Java. Exceptions are, however, not perfect. We give a more in-depth explanation of exceptions in the following slides.

Exception

Syntax of exception

```
ThrowStatement:
  throw Expression ;      throw new CardNotFound(cardName);

TryStatement:
  try Block Catches             try { ...  } catch(CardNotFound c) { ...  }
  try Block [Catches] Finally   try { ...  } catch(CardNotFound c) { ...  } finally { ...  }
```

Exception by example

```
public class CardNotFoundException extends RuntimeException {
  private String name;
  public CardNotFound(String name) { this.name = name; }
  public String toString() {
    return "The card " + name + " could not be found in the deck.";
  }
}

public Card searchCardByName(String name) {
  for (Card card : cards) { ... }
  throw new CardNotFoundException(name);
}

public void printCard(String name) {
  try {
    Card card = searchCardByName(name);
    System.out.println(card);
  }
  catch(CardNotFoundException e) {
    System.out.println(e);
  }
}
```

Advantages and disadvantages of exceptions

Advantages

- Exceptions provide a **clean way** to handle errors separately from the normal flow of the code.
- They are **non-intrusive**, meaning that the signature of the method does not need to be modified.
- Exceptions can be **arbitrarily rich** in information.

Disadvantages

- It is sometimes hard to figure out the exceptions a method can throw, documentation is therefore important for this purpose.

  See "Item 74: Document all exceptions thrown by each method", Effective Java
- Can be easily ignored, and ends up in the `main` function, which then exits and prints the exception calling stack.

Additional feature I: Checked exception

Place the exceptions a method can throw in the signature of its method:

```
public Card searchCardByName(String name) throws CardNotFoundException {
  for (Card card : cards) { ... }
  throw new CardNotFoundException(name);
}
```

This forces the calling method to treat the exception, however:

- If the exception is treated higher in the calling stack, it forces all the intermediate calling methods to add this exception to their signatures.
- It is tedious to use in practice, and not too useful.
- As suggested by *Code Clean* (Chapter 7), we will avoid using checked exceptions.
- However, it is not a universal point of view, see *"Item 70: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors", Effective Java.*

Additional feature II: try-with-resources

When acquiring a resource, such as `Scanner`, a file or a network socket, we must close it after using it:

```
public Optional<String> readFirstLineOf(String path) {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return Optional.of(br.readLine());
  } catch(IOException) {
    return Optional.empty();
  } finally {
    br.close();
  }
}
```

Additional feature II: try-with-resources

The *try-with-resources* statement is a convenient syntactic sugar to automatically closing a resource:

```
public Optional<String> readFirstLineOf(String path) {
  try (BufferedReader br =
         new BufferedReader(new FileReader(path))) {
    return Optional.of(br.readLine());
  } catch(IOException) {
    return Optional.empty();
  }
}
```

No need for the finally block, `br` is closed automatically.

Additional feature II: try-with-resources

The try-with-resources statement works with any class implementing the interface `Closeable`. For instance with `Scanner`:

```
public Optional<Integer> readInteger() {
  try (Scanner scanner = new Scanner(System.in)) {
    if(scanner.hasNextInt()) {
      return Optional.of(scanner.nextInt());
    }
  }
  return Optional.empty();
}
```

Testing

Black-box vs white-box testing

Two main categories of testing:

- **Black-box testing**: we test the functionalities of a system based on its input-output. This is how we tested *Connect Four*.
- **White-box testing**: The internal methods of the system are tested. This is (almost) how we tested *DynamicArray*.

Testing an overall behavior is generally done by black-box testing. This is also much easier to test GUI that way.

Here, we will focus on *unit testing* which is a form of white-box testing.

Why testing our project?

- **To find some bugs** before they appear in production.
- **To be the first user of our method**: a method hard to test will be hard to use.
- **To trust our code**: when we modify a part of our code, we can run the tests to verify nothing is broken.
- **To gain time**: debugging is very long and painful.

How to unit test?

As we have shown for `DynamicArray`, we do not need anything special to start unit testing. However, it might be convenient to use a dedicated testing framework, here we will use JUnit 5, aka. JUnit Jupiter. (https://junit.org/junit5/docs/current/user-guide/)

Example

You can retrieve a sample test project testing the class `DynamicArray` by doing:

```
git clone https://github.com/ptal/lab2-pokedeck/
cd lab2-pokedeck
git checkout testing
mvn test
```

Add to pom.xml

```
...
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>2.22.2</version>
        </plugin>
    </plugins>
</build>
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.7.1</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.7.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

(See https://junit.org/junit5/docs/current/user-guide/#running-tests-build-maven)

Create a JUnit test

- Create the folder `src/test/java/`.
- Inside, it can follow the same package hierarchy than in `src/main/java`, *e.g.*, `src/test/java/lab2/pcg/DeckTest.java`.
- You create one test class per class you want to test, *e.g.*, to test `DynamicArray.java`, you create the class `DynamicArrayTest.java`.

Testing a method

```
class DynamicArrayTest {
  @Test
  @DisplayName("Add elements in DynamicArray")
  void testAdd() {
    DynamicArray array = new DynamicArray();
    assertEquals(array.size(), 0);
    assertTrue(array.isEmpty());
    array.clear();
    assertEquals(array.size(), 0);
    assertTrue(array.isEmpty());
    array.add(4);
    array.add(5);
    array.add(6);
    assertEquals(array.toString(),"[4, 5, 6]");
  }
```

- `@Test`: only the methods with this annotation are called for testing.
- `assertEquals(expr1, expr2)` checks that both expressions are equal.
- `assertTrue(expr)` checks the expression is true.

BeforeEach annotation

`DynamicArrayTest` is a normal class, so we can declare attributes:

```
class DynamicArrayTests {
  private DynamicArray array;

  @BeforeEach
  void init() {
    array = new DynamicArray();
  }

  @Test
  @DisplayName("Add elements in DynamicArray")
  void testAdd() {
    assertEquals(array.size(), 0);
    ...
  }
```

Instead of declaring and initializing `array` in all testing methods, we use `@BeforeEach` to call `init()` before each method.

Testing for exceptions

```
@Test
@DisplayName("Add and remove elements in DynamicArray")
void testRemove() {
  populate(); // add the elements 4, 5, 6 in the array.
  array.remove(1);
  testArrayContent(2, "4, 6");
  array.remove(1);
  testArrayContent(1, "4");
  assertThrows(ArrayIndexOutOfBoundsException.class, () -> array.remove(1));
  array.remove(0);
  testArrayContent(0, "");
}
```

`assertThrows(E.class, () -> x.f())` tests that the method `x.f()` is throwing an exception of type `E`.

What is a good test? FIRST!

- *[F]ast*: Tests must be very fast so we run them frequently.
- *[I]solated*: Tests must not connect to a database, the network, ...
- *[R]epeatable*: Running the same test 10 times must give the same result. Randomness is proscribed.
- *[S]elf-validating*: The process of verifying if a test succeeds must be automatic, *e.g.*, we shall not need to read the output of a test.
- *[T]imely*: Don't write Java code without test, 1 method = 1 test.

Source: Pragmatic unit testing in Java 8 with JUnit

Write a good test: Right-BICEP

- *Right* Are the results right?
- *B* Are all the boundary conditions correct?
- *I* Can you check inverse relationships?
- *C* Can you cross-check results using other means?
- *E* Can you force error conditions to happen?
- *P* Are performance characteristics within bounds?

Source: Pragmatic unit testing in Java 8 with JUnit

Test Driven Development (TDD)

Methodology where the tests are central to the project:

- Instead of writing the code, then the tests, you do the opposite!
- Because we write the tests first, it forces us to think about the usability of our methods.

More about testing in Software Engineering 1 and Software Engineering 2.

Chapter IX. Parametric Polymorphism

Introduction

- **Context**: In lab 2, you implemented `DynamicArray`.
- **Problem**: It can only store integer values.
- **Today**: How can we design an array for any kind of values?

```
public class DynamicArray {
  private ?? data;

  public DynamicArray() { ?? }
  public int size() { ?? }
  public boolean add(?? e) { ?? }
  public ?? get(int index) { ?? }
}
```

Solution 1: with `Object`

We can use an array of `Object`, since, remember, every class inherits from `Object`.

```
public class ArrayList {
  static final int DEFAULT_CAPACITY = 10;
  private Object[] data;
  private int size = 0;

  public ArrayList() { data = new Object[DEFAULT_CAPACITY]; }
  public int size() { return size; }
  public void add(Object e) {
    ensureCapacity();
    data[size] = e;
    ++size;
  }
  public Object get(int i) {
    if(i < 0 || i >= size) { throw OutOfBoundException(); }
    return data[i];
  }
  private void ensureCapacity() { /* ... */ }
}
```

Problems. . .

- Make a list of String.
- Add and retrieve a string with this list.
- Is it easy and readable?

Problems...

- Make a list of String.
- Add and retrieve a string with this list.
- Is it easy and readable?

The *downcast* `(String)e;` is not very readable and secure, why?

```
ArrayList personNames = new ArrayList();
personNames.add(new String("Gertrude"));
personNames.add(new String("Johnny")));
Object e = personNames.get(1);
String name = (String)e;
```

Problems. . .

- Make a list of String.
- Add and retrieve a string with this list.
- Is it easy and readable?

The *downcast* `(String)e;` is not very readable and secure, why?

```
ArrayList personNames = new ArrayList();
personNames.add(new String("Gertrude"));
personNames.add(new String("Johnny")));
Object e = personNames.get(1);
String name = (String)e;
```

Exception `ClassCastException` for:

```
Integer i = (Integer)e;
```

And so?

- Until Java 5.0, it was the only solution.
- In Java 5.0, the concept of *generics* enables parametric polymorphism.

What are the problems of an array of `Object`?

- *Casts* are required.
- No compile-time check if the *cast* is invalid.
- For instance: `House h = (House)e`, in the previous example, compiles, but an exception is thrown at runtime.

Parametric polymorphism: don't repeat yourself!

- To avoid casts, we could create a `ArrayList` class for each types, *e.g.*, `ArrayListInteger` or `ArrayListPokemonCard`.
- But the implementation of the methods would be **redundant**.
- Actually, **we don't even need to know the underlying type** to implement these methods!
- **Solution**: Use generics!

Advantages

- The code is safer and more readable.
- Decrease runtime casts.
- Allows us to write generic classes and algorithms more easily.

Solution 2: Generics (first try)

```
public class ArrayList<T> {
  static final int DEFAULT_CAPACITY = 10;
  private T[] data;
  private int size = 0;

  public ArrayList() { data = new T[DEFAULT_CAPACITY]; }
  public int size() { return size; }
  public void add(T e) { /* as in solution 1 */ }
  public T get(int i) { /* as in solution 1 */ }
  private void ensureCapacity() { /* */ }
}
```

- `ArrayList<T>` in now parametric in a type `T`.
- `ArrayList<T>` remains a class, that can be used as a “normal class”.

A subtlety (second try)

```
public class ArrayList<T> {
  static final int DEFAULT_CAPACITY = 10;
  private T[] data;
  private int size = 0;

  public ArrayList() { data = (T[]) new Object[DEFAULT_CAPACITY];}
  public int size() { return size; }
  public void add(T e) { /* idem */ }
  public T get(int i) { /* idem */ }
  private void ensureCapacity() { /* */ }
}
```

Java does not support creating array of generic elements. Therefore, we create an array of objects that we cast immediately to the generic type.

Generics are transformed at compile-type

Backward compatible extension

- When generics were introduced, a lot of code already exists, so this existing code should not break with new Java version.
- **Solution**: Generics are *erased* at compile-time, and transformed into `Object`.
- Hence, generics are actually transformed to the code we had in solution 1, but we have additional safety guarantees.

Compiling generics

Two techniques

1. *Code expansion* (such as in C++), a new class is automatically created for each class instantiation:
   - `ArrayList<Double>` $\longrightarrow$ `ArrayListDouble`
   - `ArrayList<String>` $\longrightarrow$ `ArrayListString`
   - The parametric type `T` is replaced by the real one.
2. *Type erasure* (as in Java)
   - The parametric type `T` is replaced by a super type (`Object`).
   - Type conversions are added by the compiler automatically.
   - Generated code is the same as for solution 2.

Two usages of generic classes, a single code

- In Java, the generic type is replaced by `Object`.
- Which means that we can actually use `ArrayList` as a generic class **or not**.
- For instance, we can write `ArrayList` without generic parameter, and we will have a class with array of objects.

Two usages of generic classes, a single code

Non-generic usage

```
ArrayList x = new ArrayList();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = (String)x.get(0);
```

Two usages of generic classes, a single code

Non-generic usage

```
ArrayList x = new ArrayList();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = (String)x.get(0);
```

```
Line 2: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new String("M. George"));
Line 3: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new Integer(0));
```

Two usages of generic classes, a single code

Non-generic usage

```
ArrayList x = new ArrayList();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = (String)x.get(0);
```

```
Line 2: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new String("M. George"));
Line 3: warning: [unchecked] unchecked call to add(T) as a member
    of the raw type ArrayList
 x.add(new Integer(0));
```

- Compiler will output *warnings*.
- Heterogeneous array (several types) are generally a bad idea, it is better to use inheritance or enumeration instead.
- Always the risk to generate an exception if we mess up the cast.

Two usages of generic classes, a single code

Generic usage

```
ArrayList<String> x = new ArrayList<String>();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = x.get(0);
```

Two usages of generic classes, a single code

Generic usage

```
ArrayList<String> x = new ArrayList<String>();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = x.get(0);
```

```
Line 3: error: incompatible types: Integer cannot be converted
               to String
    x.add(new Integer(0));
```

Two usages of generic classes, a single code

Generic usage

```
ArrayList<String> x = new ArrayList<String>();
x.add(new String("M. George"));
x.add(new Integer(0));
String name = x.get(0);
```

```
Line 3: error: incompatible types: Integer cannot be converted
               to String
    x.add(new Integer(0));
```

- The compiler generates an *error*.
- It guarantees we can only put in the list what is specified in the angle brackets (`ArrayList<MyType>`).
- No need to *cast* when we use `get`, we give the compiler enough information so it can safely add the cast itself.

Advanced concepts of generics

Multiple generics parameters

- Some classes need several generics type.
- For instance in the associative array data structure.

Associative array

- Associate a key to a value. For instance, the name of someone to its address.
- `HashMap<String, Address> directory = new HashMap<String, Address>();`

```
public class SimpleMap<K,V> { // Key and Value
  private ArrayList<Pair<K,V>> data;
  private static class Pair<K,V> {
    public K key;
    public V value;
  }
  // ...
}
```

Type inference

- Type inference allows us to ask the compiler to *guess* (or *infer*) the type of an expression.
- It is not very powerful in Java but still useful for clarity.

```
HashMap<String, Address> directory = new HashMap<>();
```

Generic methods I

Challenge

Create a static method `head` which takes an `ArrayList` and returns the first element.

Generic methods I

Challenge

Create a static method `head` which takes an `ArrayList` and returns the first element.

Non-generic

```
public class ArrayListTools {
  public static Object head(ArrayList data) {
    return data.get(0);
  }}
```

Generic methods II

Is it working if we write the following?

```
ArrayList<String> names = new ArrayList<String>();
ArrayListTools.head(names);
```

Generic methods II

Is it working if we write the following?

```
ArrayList<String> names = new ArrayList<String>();
ArrayListTools.head(names);
```

- No! `ArrayList<String>` does not inherit from `ArrayList<Object>`.
- *Invariant types*: Inheritance is not propagated to type parameters, *i.e.*, `X<T>` and `X<U>` are never subtypes of each other.
- *Covariant types*: This is not the case with array, *i.e.*, `String[]` is a subtype of `Object[]`.

Generic methods II

Is it working if we write the following?

```
ArrayList<String> names = new ArrayList<String>();
ArrayListTools.head(names);
```

- No! `ArrayList<String>` does not inherit from `ArrayList<Object>`.
- *Invariant types*: Inheritance is not propagated to type parameters, *i.e.*, `X<T>` and `X<U>` are never subtypes of each other.
- *Covariant types*: This is not the case with array, *i.e.*, `String[]` is a subtype of `Object[]`.

We should use a generic method:

Generic method

```
public class ArrayListTools {
  public static <T> T head(ArrayList<T> data) {
    return data.get(0);
  }}
```

Bounded type parameters

When a class is instantiated with a generic type `T`, it has no information on `T`, thus cannot call any method on this object.

We can bound the type.

```
class SortedArrayList<T extends Comparable> {
  private T[] data;
  // ...
    data[i].compareTo(data[i+1]); // ok, T implements Comparable.
}
```

- Subtlety: We use `extends` even if `Comparable` is an interface.
- We can also give several type bounds: `<T extends Comparable & Cloneable>`.

More on generics

- Lower and upper type bounds.
- Wildcard (<?>).
- ...

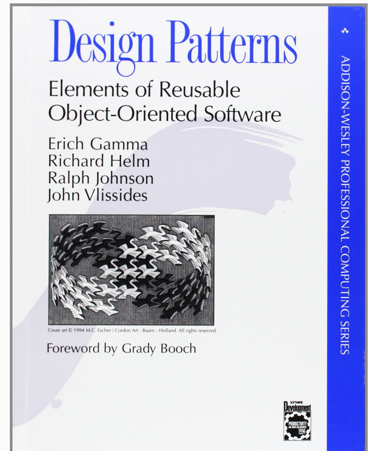
More on the topic:

- *Effective Java, Chapter 5.*
- http://en.wikipedia.org/wiki/Generics_in_Java
- http://en.wikipedia.org/wiki/Wildcard_%28Java%29
- On a more general topic: http://en.wikipedia.org/wiki/Covariance_and_contravariance_%28computer_science%29
- Another book: *Java Generics and Collections, Maurice Naftalin and Philip Wadler, O'reilly, 2006*

Chapter X. Design Pattern

Design pattern

- A design pattern is a **reusable** general solution to a software problem.
- A way to organise the code to increase flexibility, reusability, maintainability, ...
- Generally based on inheritance, subtype polymorphism, and interfaces.



Why are design patterns interesting?

- Introduce a common vocabulary among developers: make it easier to understand the code.
- They are robust solutions, designed over the years by expert developers.
- Extensible and modular: weak coupling between software components.

Classification of design patterns

1. **Creational patterns**: to build an object when it is complicated (e.g., to "help" the constructor).
   - *Factory, AbstractFactory, Builder, ...*
   - `ASCIIBattlefieldBuilder` builds `Battlefield`.
2. **Structural patterns**: to extend a class with functionalities without modifying it.
   - *Adapter, Facade, Decorator, Proxy, Composite, ...*
3. **Behavioral patterns**: to introspect an object and/or customized its behavior.
   - *Iterator, Observer, Strategy, Visitor, ...*
   - `TileVisitor` allows us to visit the tiles of the battlefield.

A selection of design patterns

We discuss five design patterns:

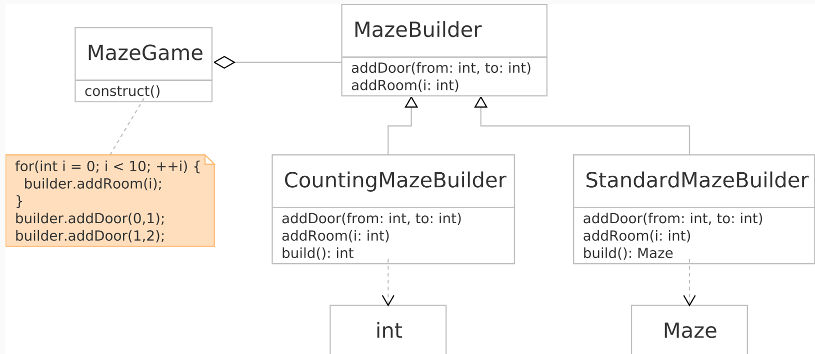
1. *Builder pattern*: used in LOL 2D.
2. *Composite pattern*: used in Calculator and MC (lab 4).
3. *Facade pattern*: used in LOL 2D.
4. *Visitor pattern*: used in LOL 2D.
5. *Observer pattern*: should be used in LOL 2D.

Builder Design Pattern

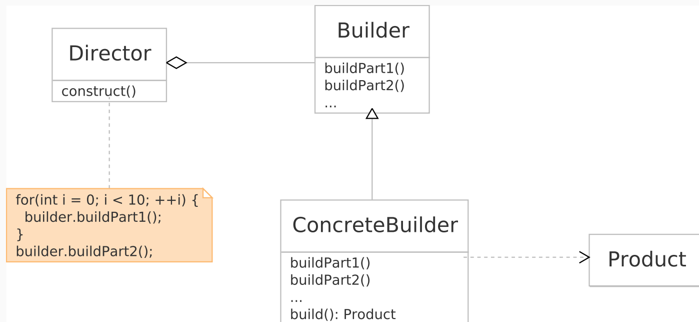
Builder Pattern: Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

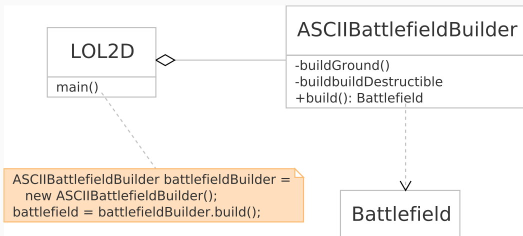
Motivation: Maze Builder



General case: Builder design pattern



A restricted usage in LOL 2D



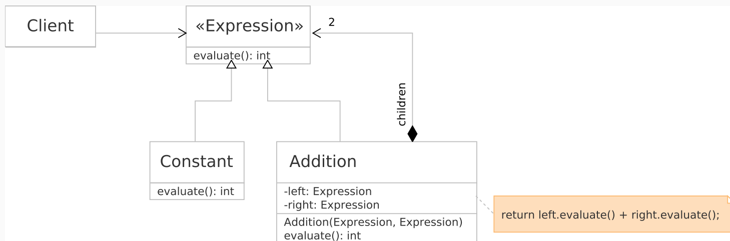
- Constructing the battlefield with an ASCII file is 100 LOC.
- Usage of the builder to separate object construction from the object itself.
- Currently, no need for a `Builder` interface.
- Could be added later when required, *e.g.*, suppose you want to propose a map editor.

Composite Design Pattern

Intent

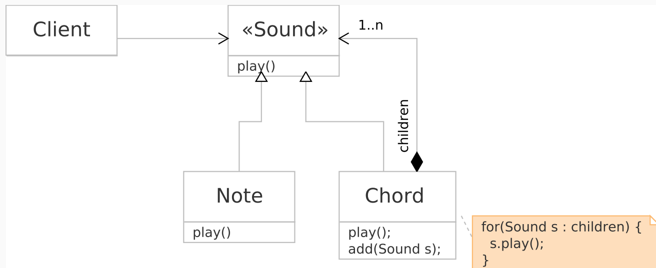
Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation: Calculator



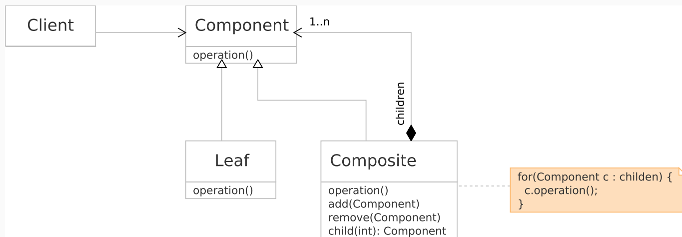
A constant or a *composition* of constants through `Addition` are manipulated uniformly through `Expression`.

Motivation: Musical score



A note or a *composition* of notes through `Chord` are manipulated uniformly through `Sound`.

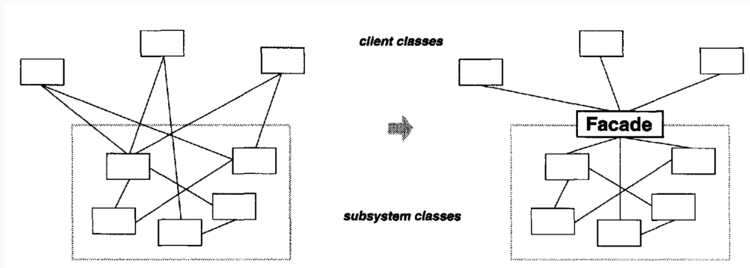
General case: Composite design pattern



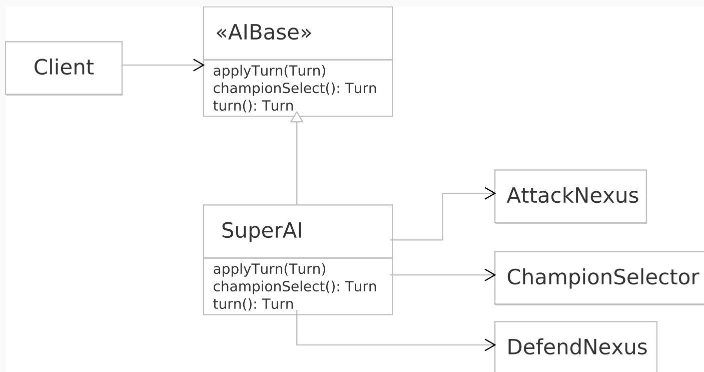
Facade Design Pattern

Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.



Motivation: AI interface



- Each client can implement its own AI, which might be super sophisticated and involves many components.
- All AIs are used in Client the same way, through the facade `AIBase` interface.

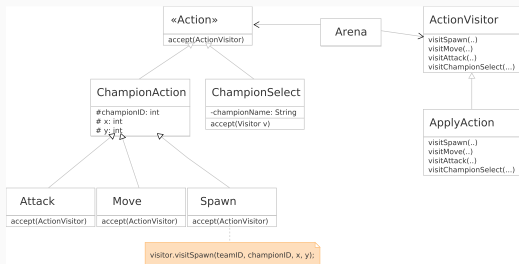
Visitor Design Pattern

Intent

> *Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.*

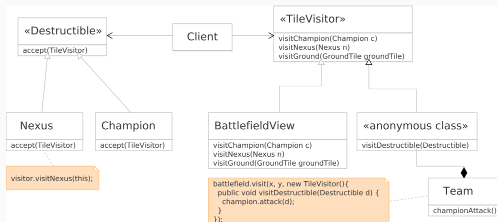
It is a solution to the *Expression problem* mentioned in Live coding 4.

Motivation: Action on battlefield



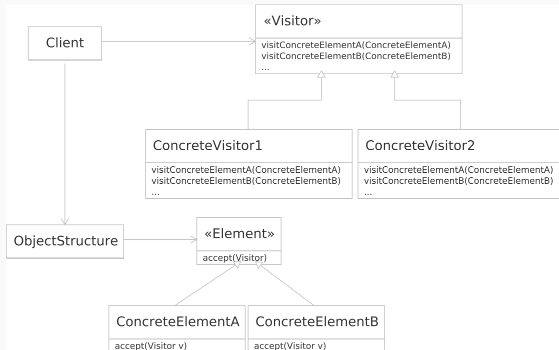
- An action has an effect on the battlefield (e.g., moving a champion, attacking a destructible, ...).
- The class `Turn` has an `ArrayList<Action>`.
- How to iterate over the list of actions, and know the concrete subtype?
- The visitor pattern allows us to introspect the actions.

Motivation: Tiles of the battlefield



- The battlefield is constituted of different kind of tiles, either ground or destructible.
- The visitor allows us to introspect a destructible tile.

General case: Visitor design pattern

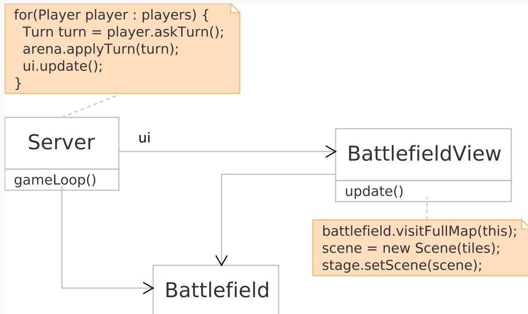


Observer Design Pattern

Intent

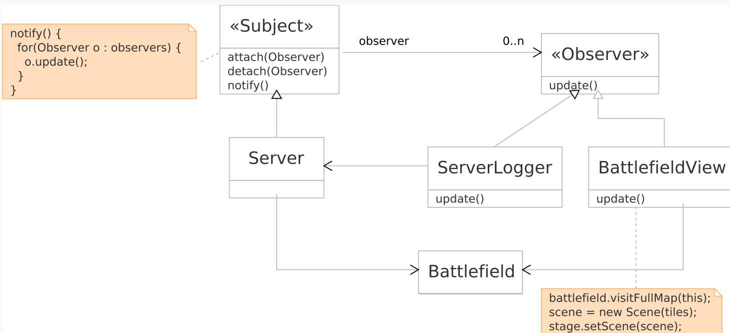
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Motivation: Server / UI communication



Currently, the server directly communicates to the UI.

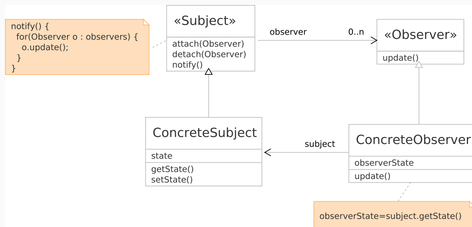
Motivation: Server / UI observer



Observer pattern in Java

In Java, the interface `Observer` and the class `Observable` (`Subject` in the example) are already provided!

General case: Observer design pattern



Chapter XI. Modern Java Concepts

A Tour of Modern Java Concepts

Programming languages are not static entities, but evolve over the years by incorporating new concepts:

- Java 8: lambda expressions, unsigned integer arithmetic, ...
- Java 9: modules, ...
- Java 10: local-variable type inference:

```
var list = new ArrayList<String>();
// equivalent to
ArrayList<String> list = new ArrayList<String>();
```

- Java 14: switch expressions

```
static void howMany(int k) {
    System.out.println(
        switch (k) {
            case  1 -> "one";
            case  2 -> "two";
            default -> "many";
        }
    );
```

A Tour of Modern Java Concepts

- Java 15: text block, new garbage collectors, ...
- Java 16: pattern matching for `instanceof`, records:

```
record PokemonCard(String name, String description, int hp, int level) {}

PokemonCard c = new PokemonCard("Pikachu", "Great Pokemon", 100, 1);
System.out.println(c.name()); // accessors automatically generated.
PokemonCard c2 = ...;
if(c.equals(c2)) { ... }  // equals automatically generated.
```

The constructors, accessors, methods `equals`, `hashCode`, `toString`, ... are automatically generated.

Limitation

Records should solely be used when we need to store "data" and no interesting treatment is performed on those. They are **immutable** and cannot inherit from classes or implement interfaces.

A Tour of Modern Java Concepts

We explain in more depth two advanced concepts of Java:

- Nested classes
- Lambda expressions

Nested classes

Nested classes

For now, we have only used classes declared at “top-level”.
A class can also be declared inside other Java entities:

- *Inner class*: a class within a class with access to the containing class's fields and methods.
- *Static nested class*: a class within a class *without* access to the containing class's *non-static* fields and methods.
- *Local class*: a class declared within a method.
- *Anonymous class*: a class without a name.

See also *Effective Java. Item 24: Favor static member classes over nonstatic*.

Inner class

```java
public class Arena {
  private ArrayList<Team> teams;
  private Battlefield battlefield;

  private class ApplyAction implements ActionVisitor {
    ArrayList<Integer> championsWhoActed;

    public void visitSpawn(int teamID, int championID, int x, int y) {
      teams.get(teamID).spawnChampion(championID, x, y));
    }
    ...
  }
}
```

- `ApplyAction` has access to all private fields of `Arena`.
- `ApplyAction` can be used like a normal class inside `Arena`.
- As `ApplyAction` is `private`, it is not visible outside of `Arena`.
- `Arena` cannot access the private members of `ApplyAction`.

Inner class

How does it work?

An instance of an inner class has a *hidden field* containing the reference of an instance of the containing class.

```
public class Arena {
  private class ApplyAction implements ActionVisitor {
    Arena arena;
    public ApplyAction(Arena arena) {
      this.arena = arena;
    }
    public void visitSpawn(int teamID, int championID, int x, int y) {
      arena.teams.get(teamID).spawnChampion(championID, x, y));
    }
    ...
  }
```

Therefore, you can only instantiate an inner class in the context of its containing class.

Static nested class

```
public class Battlefield {
  public static enum GroundTile {
    GRASS,
    ROCK;
    // ...
  }
}
```

- Very similar to a class declared “normally”, but in addition can access private static members of the containing class.
- It is a way to indicate a class is very dependent w.r.t. another one.
- Here `GroundTile` existence depends on `Battlefield`.
- However, it is often better to use package to group classes together, so static nested classes have a limited usage.

Local class

Perhaps surprisingly, you can declare a class almost anywhere a local variable can be declared:

```
public class NumberValidator {
  public static void validatePhoneNumber(String phoneNumber) {
    class PhoneNumber {
        String formattedPhoneNumber;
        PhoneNumber(String phoneNumber){...}
        public String getNumber() {
            return formattedPhoneNumber;
        }
    }
    PhoneNumber myNumber1 = new PhoneNumber(phoneNumber);
    if (myNumber1.getNumber() == null) { ... }
    ..
  }
}
```

This kind of nested class is not very useful and used.

Anonymous class

The last kind of nested class, anonymous class, is very common.

```
public class Arena {
  // ...
  @Override public String toString() {
    StringBuilder map = new StringBuilder();
    visitFullMap(new TileVisitor() {
      @Override public void visitGround(GroundTile groundTile, int x, int y) {
        map.append(GroundTile.stringOf(groundTile));
        newline(x);
      }

      @Override public void visitChampion(Champion c) {
        map.append('C');
        newline(c.x());
      }
      // ...
    });
    return map.toString();
  }
```

Anonymous class

- It is similar to a local class, but without a name.
- Can access and modify local objects (*e.g.*, `map` in the previous example).
- Useful when a class is local to a method, only need to be instantiated in one place, and is not reusable outside of the current context.

Limitations

- Cannot implement multiple interfaces.
- Cannot be used inside expression relying on type name such as `instanceof`.
- Clients of anonymous class can only call the methods of the super types (which can be overridden by the anonymous class).

Nested classes: rules of thumb

- Should be kept short.
- Use static nested classes if you don't need a reference to an object of the containing class.
- Use lambda expressions instead of anonymous classes when it is possible.

Lambda Expressions

Lambda expressions

Lambda expressions are inspired by the *functional programming* paradigm (that you will learn in PF3).

Functional programming

- **Immutable memory.**
- **First-order functions**: we can pass functions to functions as arguments, and functions can be returned too!

Lambda expressions are first-order functions over mutable memory.

The following slides are based on *Effective Java. Chapter 7: Lambdas and Streams* and `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

Motivation

We take the example of Pokedeck (lab 2), where we have a collection of cards that we must search. We can search a card by ID, by name, by card type:

```
public class Deck {
  private ArrayList<Card> deck;
  // ...
  public ArrayList<Card> getCardsByName(String name) {
    ArrayList<Card> searchResults = new ArrayList<Card>();
    for (Card card : deck) {
      if(name.equals(card.getName())) {
        searchResults.add(card);
      }
    }
    return searchResults;
  }
  public ArrayList<Card> getCardsById(String id) {
    ArrayList<Card> searchResults = new ArrayList<Card>();
    for (Card card : deck) {
      if(id.equals(card.getId())) {
        searchResults.add(card);
      }
    }
    return searchResults;
  }
  public ArrayList<Card> getCardsByCardType(CardType cardType) {...}
```

Motivation

- We can observe a recurring pattern.
- All these methods look the same: don't repeat yourself (DRY) principle!

Two solutions

1. The old one with interfaces and anonymous classes.
2. The new and cool one with lambda expressions.

Solution 1: Interface and anonymous class

First, we create a `DeckFilter` interface:

```
interface DeckFilter {
  boolean keepCard(Card c);
}
```

We can then code a generic search method using this filter:

```
public ArrayList<Card> search(DeckFilter filter) {
  ArrayList<Card> searchResults = new ArrayList<Card>();
  for (Card card : deck) {
    if(filter.keepCard(card)) {
      searchResults.add(card);
    }
  }
  return searchResults;
}
```

Solution 1: Interface and anonymous class

The user of the class `Deck` can choose its own search criterion:

```
Deck deck = ...;
deck.search(new DeckFilter() {
  @Override boolean keepCard(Card card) {
    return name.equals(card.getName());
  }
})
```

As a bonus, we also made the class more respectful of the open-closed principle!

Solution 1: Interface and anonymous class

Note that we can provide a number of default filters:

```
public class DeckSearch {
  public static class FilterByName implements DeckFilter {
    private String name;
    public FilterByName(String name) {
      this.name = name;
    }
    @Override boolean keepCard(Card card) {
      return name.equals(card.getName());
    }
  }
  public static class FilterByID implements DeckFilter { ... }
}
```

That can be used in client code:

```
Deck deck = ...;
deck.search(new DeckSearch.FilterByName(name));
```

Drawbacks of solution 1

- It introduces a lot of "boilerplate code": interfaces, static nested classes.
- And it is not always very readable: anonymous classes.

Lambda expressions provide a more satisfying way to solve this problem.

Solution 2: using lambda expressions

The interface `DeckFilter` is what is called a **functional interface** because it has only one abstract method:

```
@FunctionalInterface interface DeckFilter {
  boolean keepCard(Card c);
}
```

As a syntactic shortcut, we can use *lambda expressions* to implement this interface:

```
Deck deck = ...;
deck.search(card -> name.equals(card.getName()));
```

The code `card -> name.equals(card.getName())` is a lambda expression.

Lambda expressions

Another example: sorting a list of strings by length:

```
Collections.sort(words,
  (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

- We declare a function taking two arguments `s1` and `s2`.
- Implicit `return` keyword.
- All types are automatically inferred by the compiler.

Syntax of lambda expressions

```
(arg1, arg2, ...) -> A Java expression
(arg1, arg2, ...) -> {
  One or more statements (possibly ending with a return statement)
}
```

Improving solution 2 using streams

Streams encapsulate a number of generic and common operations on list: `filter`, `map`, `reduce`, `sorted`, ...

Lambda expressions really shine in cooperation with streams:

```
public List<Card> search(DeckFilter filter) {
  return deck.stream()
    .filter(filter)
    .collect(Collectors.toList());
}
```

We create a stream, filter on this stream and then collect the result.

Compare with the previous solution: it is much shorter.

Improving (again) solution 2 using standard functional interface

The interface `DeckFilter` contains a single filtering method which is actually quite common. Java defines a number of standard interface so we do not need to redefine ours:

```
public List<Card> search(Predicate<Card> keepCard) {
  return deck.stream()
    .filter(keepCard)
    .collect(Collectors.toList());
}
```

`Predicate<Card>` contains a generic method `boolean test(T t)`. The call to `search` stays the same, as the lambda expression automatically implements `Predicate<Card>`:

```
deck.search(card -> name.equals(card.getName()));
```

Standard functional interface

Functional interfaces already present in `java.util.function`:

Interface	Function Signature	Example
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

Method references

A (static) method can also be used where a lambda expression is expected:

Method Ref Type	Example	Lambda Equivalent
Static	`Integer::parseInt`	`str -> Integer.parseInt(str)`
Bound	`Instant.now()::isAfter`	`Instant then = Instant.now();` `t -> then.isAfter(t)`
Unbound	`String::toLowerCase`	`str -> str.toLowerCase()`
Class Constructor	`TreeMap<K,V>::new`	`() -> new TreeMap<K,V>`
Array Constructor	`int[]::new`	`len -> new int[len]`

Summary

Method references often provide a more succinct alternative to lambdas. Where method references are shorter and clearer, use them; where they aren't, stick with lambdas.

A more complete example using lambda and streams

Here a function retrieving the sorted list of ID of all cards fulfilling a criterion:

```
public List<Integer> filteredSortedID(Predicate<Card> keepCard) {
  return deck.stream()
    .filter(keepCard)
    .map(Card::getID)
    .sorted()
    .collect(Collectors.toList());
}
```

- *Fluent interface*: we can chain the operations.
- *Lazily evaluated*: the whole thing is only evaluated when we arrive on `collect`. It means the collection is not traversed more than once in case of multiple `map`/`filter` operations!

Chapter XII. Network Programming in Java

Introduction

Today, we learn about networking in Java by implementing a chatting app!

Implementing Discord: step by step

1. Discord V1: 1 client and 1 server (echo server).
2. Discord V2: n clients and 1 server, but the clients cannot see the messages of others.
3. Discord V3: n clients and 1 server, the messages are broadcasted, and the server can be shutdown.

Please clone the following repository:
`https://github.com/ptal/chatroom`

Discord V1: 1 server - 1 client

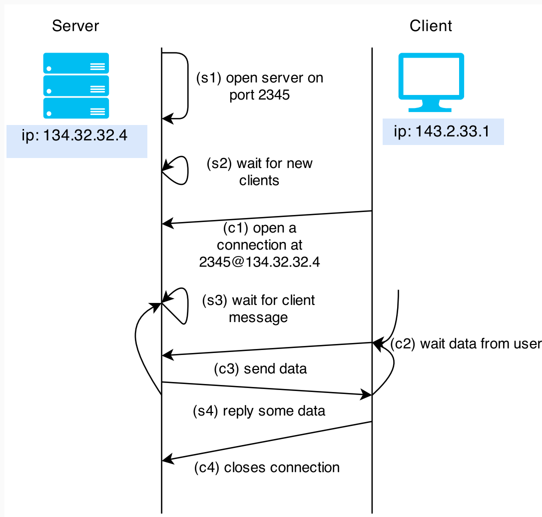
Networking in a nutshell

- Each machine is identified by an IP address.
- To communicate with a machine, we open a communication channel on a particular port (*e.g.*, 80 for `http`).
- Ports numbered from 0 to 1023 are reserved for common protocols (http, dns, echo, ...).

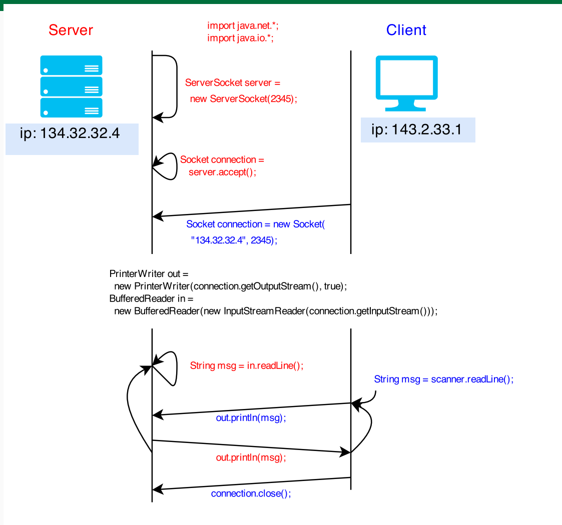
Client-server model

- A server listens the request of the clients on a particular port.
- A client connects to the server with the coordinate (ip, port).
- The server can act as an intermediate among clients (*e.g.*, chatting app).
- It is a centralized model because the server is at the center and all communications go through the server.
- When the server is dead, nobody can communicate anymore (in contrast to peer-to-peer network).

Client-server communication scenario



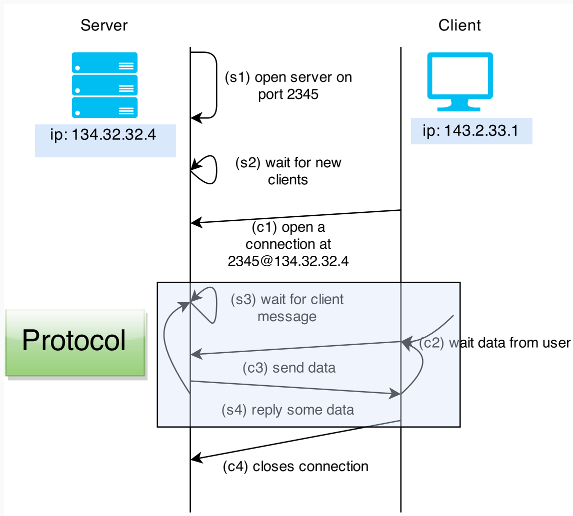
Client-server in Java



Implement this scenario in `Server.java` and `Client.java`

Networking Protocol

Protocol



What is a protocol

- A protocol specifies how the server and clients communicate.
- Basically, who send what at what time.
- If the protocol is well-documented, we can implement a client without looking at the code of the server.

Example

1. The client sends a pseudo and a password
2. The server verifies if it is correct and send `ok` if it is, and `ko` otherwise.
3. The client go to step (1) if it receieves `ko`. Otherwise, it continues by asking profile information.
4. The server send the information.
5. ...

Protocol format

- Two families
  1. Binary: Data is structured and interpreted following the size in bytes of the different fields.
  2. Text: Data is an array of characters, possibly describing a high-level format (*e.g.*, XML, JSON).

Binary format procotol

For instance, network protocols are specified in a binary format.

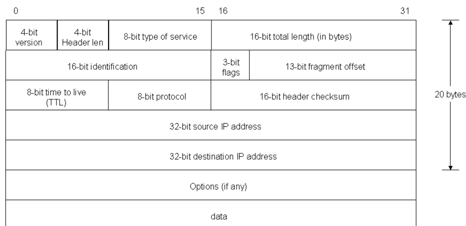


Image from `http://iacs.seas.harvard.edu/courses/ac263/course/protocols.html`

Binary format procotol

- Advantage of binary format is that the size of a network packet is minimized.
- However, packet are not easily readable, harder to implement and not adequate for interoperability.
- **Normally, only use binary format if the text format was shown to be too slow**.
- An exception: serialization...

A useful binary protocol: Serialization

Serialization is the process of turning a data structure, in our case a Java object, into a sequence of bytes. The sequence of bytes can be written in a file or transmitted over the network.

- (+) Completely automatic and transparent for us.
- (+) Very easy to use (`implements Serializable` in Java).
- (-) Not interoperable: only for the communication between 2 Java programs.

Example from game/action/Turn.java in LOL2D

```
public class Turn implements Serializable {
  public void send(Socket socket) throws IOException {
    OutputStream outputStream = socket.getOutputStream();
    ObjectOutputStream objectOutputStream = new ObjectOutputStream(outputStream);
    objectOutputStream.writeObject(this);
  }

  @SuppressWarnings("unchecked")
  public static Turn receive(Socket socket) throws IOException {
    InputStream inputStream = socket.getInputStream();
    ObjectInputStream objectInputStream = new ObjectInputStream(inputStream);
    Object rawTurn = null;
    try { rawTurn = objectInputStream.readObject(); } catch(Exception e) {}
    if(!(rawTurn instanceof Turn)) {
      throw new BadProtocolException("turn of type ‘Turn‘.");
    }
    return (Turn) rawTurn;
  }
}
```

Text-based protocol

Example from the *add-ons* server of the game *Battle for Wesnoth* (`http://hyc.io/wesnoth/umcd.pdf`).

Request to delete an add-on

- Format:

```
[request_umc_delete]
  id = ID
  password = PASSWORD
[/request_umc_delete]
```

- Fields description:

  ID The ID of the UMC we want to delete.

  PASSWORD The password of the UMC.

Text-based protocol

Reply from the server

- An error packet can be sent for the common reasons (see 2.4.2) but also because:
  1. The password is wrong.
- In case of success, a packet with no field is sent.

```
[request_umc_delete]
[/request_umc_delete]
```

Optional exercise: JSON protocol

- Encapsulate a message in a JSON packet.
- For this purpose, specify a very simple protocol.

Exemple

```
{
  name: "request_umc_delete",
  id: 132,
  password: "UTE6542162143ECUSACE"
}
```

Example of JSON specification: `https://github.com/ptal/online-broker/wiki/Online-broker-API`

JSON Library

Maven dependency (to add in pom.xml)

```
<dependency>
  <groupId>org.json</groupId>
  <artifactId>json</artifactId>
  <version>20141113</version>
</dependency>
```

Example

```
import org.json.simple.JSONObject;
//...
JSONObject obj = new JSONObject();
obj.put("name","request_umc_delete");
obj.put("id", new Integer(132));
obj.put("password", "UTE6542162143ECUSACE");
StringWriter out = new StringWriter();
obj.writeJSONString(out);
String jsonText = out.toString();
JSONObject sameObj = new JSONObject(jsonText);
```

Discord V2: 1 server - n isolated clients

Multi-clients server

Challenge

- How can a server manages several clients simultaneously?
- We would like to perform several concurrent actions:
  1. Accept new clients.
  2. Wait messages from clients already connected.
- The problem is that these two actions are *blocking*, we can do one or the other.

Solution 1: two steps protocol

- The easiest solution is to wait for a number of clients and then start the discussion.
- Each client talks one after the other.
- This is what happens in LOL 2D.
- But not very useful for a chatroom...

Solution 2: N+1 programs

The intuition is to have:

- 1 program accepting new clients.
- N programs communicating with the N clients connected.

Generating so many programs is heavy for the systems and consume a lot of resources. A solution is to use *threads*.

Threads

There exists two ways to create *threads* in Java: inheriting from `Thread` or implementing the interface `Runnable`.

```
class Connection extends Thread {
  Socket socket;
  Connection(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    // code communicating with the client
    ...
    socket.close();
  }
}
...
Connection connection = new Connection(socket);
connection.start();
```

Runnable

If your class need to inherit from something else, you can use the interface `Runnable`:

```
class Connection implements Runnable {
  Socket socket;
  Connection(Socket socket) {
    this.socket = socket;
  }

  public void run() {
    // code communicating with the client
    ...
    socket.close();
  }
}
```

Exercise: Discord V2

Create an instance of the Connection class each time the server receives a new request:

```
while(true) {
  Socket socket = server.accept();
  System.out.println("New client at " + socket);
  new Connection(socket).start();
}
```

Discord V3: 1 server - n clients (+ clean shutdown of the server)

Clean shutdown of the server

To stop the server, you must signal to all running threads that you want to stop.

```
class Connection extends Thread {
  ...
  public void interrupt () {
    super. interrupt ();
    try {
      socket . close ();
    } catch (IOException e) {} // quietly close
  }
  public void run() {
    try {
      ...
    }
    catch (InterruptedIOException e) {
      Thread.currentThread(). interrupt ();
    }
    catch (IOException e) {
    }
    socket . close ();
  }
}
...
Connection connection = new Connection(socket);
...
connection. interrupt ();
```

Clean shutdown of the server

- To stop all connections, you must first register those in an array.
- Then, the method `join` of a thread allows us to wait for the end of the thread execution.

```
ArrayList<Connection> connections = new ArrayList<Connection>();
...
for(Connection c : connections) {
  c.interrupt();
}
for(Connection c : connections) {
  c.join();
}
```

Discord V3: Chat room

- Each time the server receives a message, it is broadcasted to all connected clients.
- We keep all the connections in the `Server` class.
- Each time a client is accepted, it is added in the list `room`, and when it quits, it is removed.

```
class Server {
  ArrayList<Connection> room;
  ...
  public void broadcast_msg(String msg) {
    for(Connection c : connections) {
      c.send(msg);
    }
  }
}
```

Two threads for the client

Since the client can send and receive messages, we need one thread for each:

```
class Client implements Runnable {
  MessageReader msgReader;
  public Client(...) {
    msgReader = new MessageReader(in);
    msgReader.start();
  }

  public void run() {
    while ((userInput = stdin.nextLine()) != null) {
      out.println(userInput);
    }
  }
}
```

Two threads for the client

Since the client can send and receive messages, we need one thread for each:

```
class Client implements Runnable {
  MessageReader msgReader;
  public Client(...) {
    msgReader = new MessageReader(in);
    msgReader.start();
  }

  public void run() {
    while ((userInput = stdin.nextLine()) != null) {
      out.println(userInput);
    }
  }
}
```

Improve this code so the program exits when the user types "\quit".

Quick Notes on Multithreading

Race conditions

- To communicate, *threads* share memory (*e.g.*, they share an object).
- This communication model, called *shared memory multithreading* is very hard to use right.
- Indeed, two threads can write in the same variable at the same time.

Example

Let x, y be shared and initialized to 0.

Thread 1	**Thread 2**
$\mathrm{x} = 1$	$\mathrm{y} = 1$
$\mathrm{r1} = \mathrm{y}$	$\mathrm{r2} = \mathrm{x}$

What are the possible results?

Race conditions

- To communicate, *threads* share memory (*e.g.*, they share an object).
- This communication model, called *shared memory multithreading* is very hard to use right.
- Indeed, two threads can write in the same variable at the same time.

Example

Let x, y be shared and initialized to 0.

Thread 1	Thread 2
$\mathrm{x} = 1$	$\mathrm{y} = 1$
$\mathrm{r1} = \mathrm{y}$	$\mathrm{r2} = \mathrm{x}$

What are the possible results?

Everything is possible: `r1=1,r2=1` or `r1=1,r2=0` or `r1=0,r2=1` but also `r1=0,r2=0`.

Synchronized

A race condition occurs when two threads write on the same variable. How to retreive some sequentiality and force the threads to write one at a time?

```
public class SafeInteger {
  private int x = 0;

  public synchronized void increment() {
    x = x + 1;
  }
}
```

The keyword `synchronized` guarantees that only one thread can only enter a method at a time.

Back to the server

- We must carefully add `synchronized` at the right places.
- What are the resources shared by the different *threads*?
- Mainly the list of connections and during the *broadcast*.
- Moreover, we do not want to keep the arrival order of the messages of the clients.

Exercise: improve the server to erase a client from the connections list when it disconnects or sends "\quit".

Concurrency vs Parallelism

From `http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html`:

- *Parallelism*: A condition that arises when at least two threads are executing simultaneously.
- *Concurrency*: A condition that exists when at least two threads are making progress. A more generalized form of parallelism that can include time-slicing as a form of virtual parallelism.

Conclusion

Multithreading is hard and generally unsafe, avoid to use it as much as you can.

We will discuss about various parallel programming models in PF3.

See also *The problem with threads, Lee Edward, 2006*