Java for Game AI
1: Java Basics

Raluca D. Gaina – r.d.gaina@qmul.ac.uk

http://gameai.eecs.qmul.ac.uk

Queen Mary University of London

Outline

- Object-Oriented Programming
- Types, Variables, Methods
- Classes, Inheritance and Interfaces
- Java Coding

1: Java Basics

Object-Oriented Programming

OOP

- Organizes software design around data, rather than functions and logic. Data -> **classes** = objects with different **properties** (**variables**, or fields) and **behaviours** (**methods**, or functions).

Bomb

Properties	Behaviours
bombID ownerID timeToLive position velocity blastStrength	explode() tick()

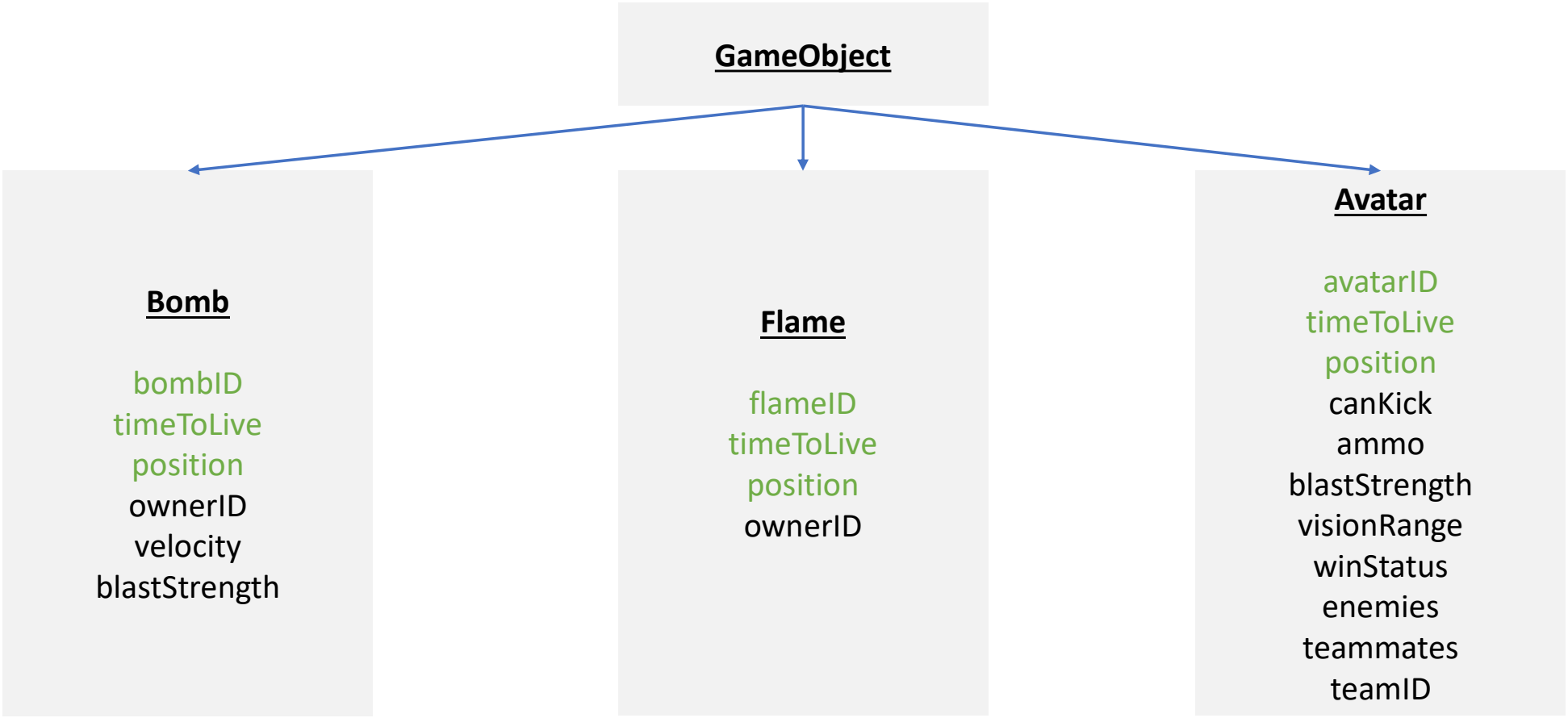
```
bombID1, ownerID1, timeToLive1, blastStrength1, position1, velocity1;
bombID2, ownerID2, timeToLive2, blastStrength2, position2, velocity2;
.....

500 bombs? -> too many variables!
```

- Why use classes?

```
Bomb bomb1 = new Bomb();
Bomb bomb2 = new Bomb();
bomb1.timeToLive -= 1;
```

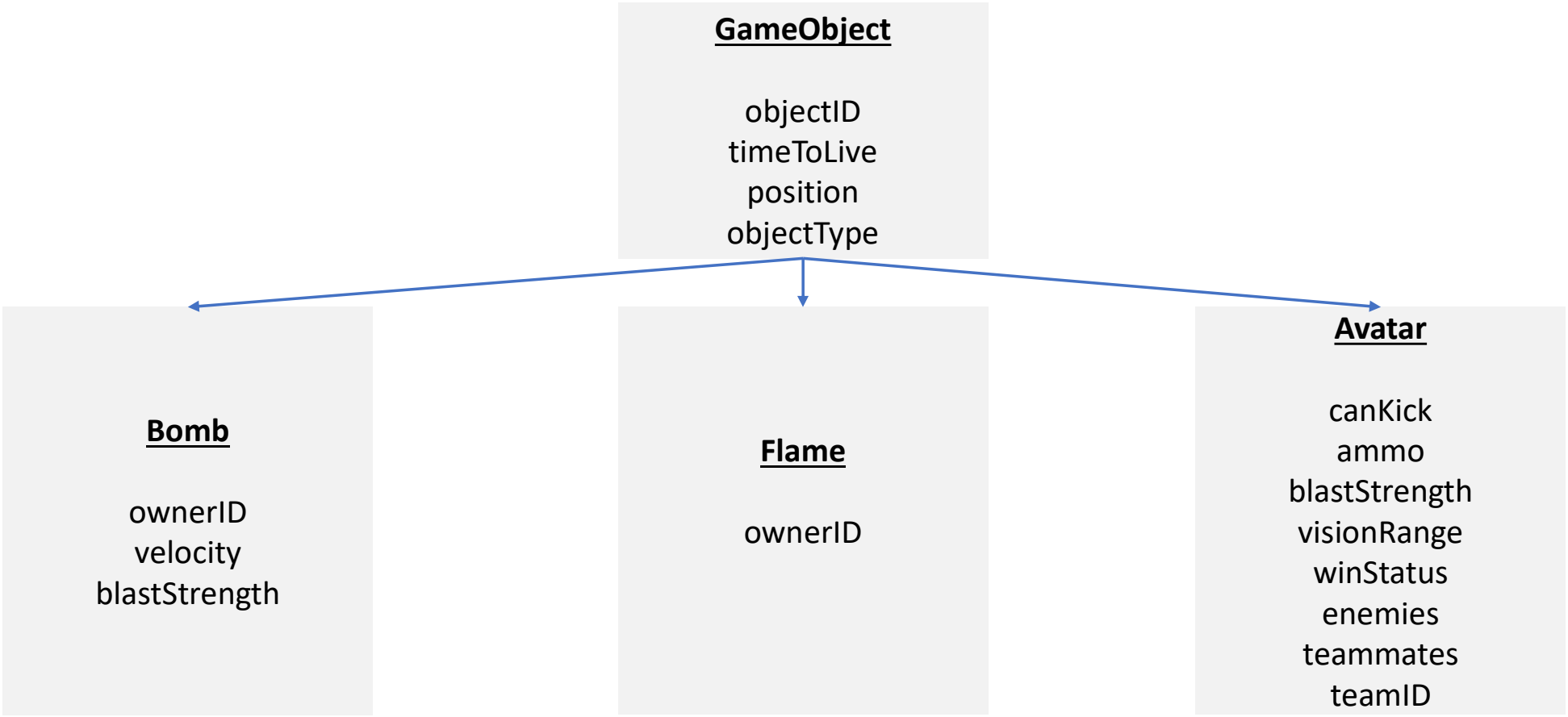
OOP

- But Bomb is just one type of game object:



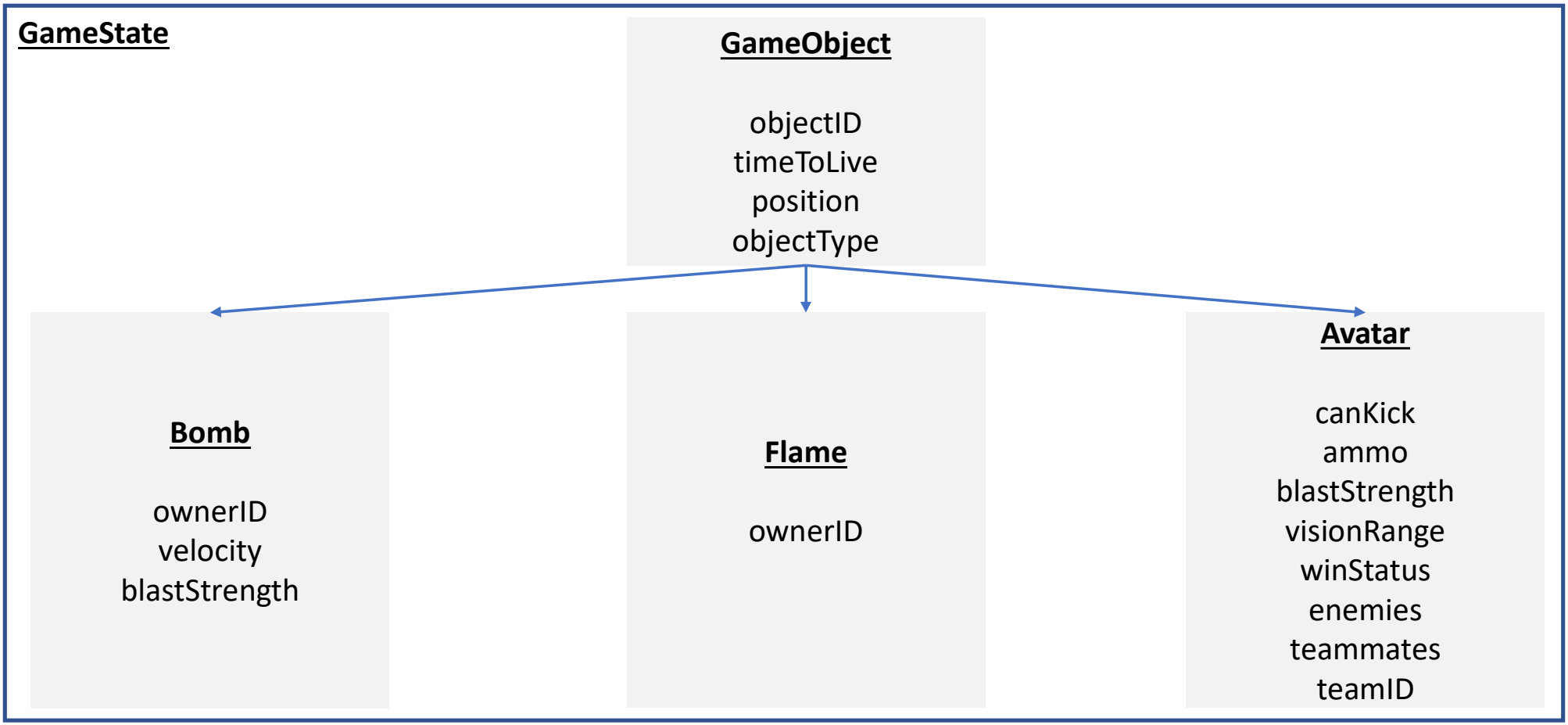
OOP

- But Bomb is just one type of game object:



OOP

- And all objects are part of our game state, which is also an object with its own properties:



OOP

- And all objects are part of our game state, which is also an object with its own properties:

GameState

bombs: list of Bomb objects
flames: list of Flame objects
avatars: list of Avatar objects
powerUps: list of GameObject objects of type PowerUp
board: matrix of game object types
…

- In *Pommerman*, some of these variables are actually in the **ForwardModel** object instead, and an instance of this object is kept within the **GameState** object.
- OOP is flexible to allow defining
  - Objects within other objects
  - Objects with common properties
  - Objects with common functionality

OOP - Principles

- **Encapsulation**: Each object, its properties (and current state) and behaviours are privately held within a class. Other objects can only modify this class if given permission to, and access to both properties and behaviours can be strictly controlled. This is called *data hiding* and offers increased security and avoids data corruption.

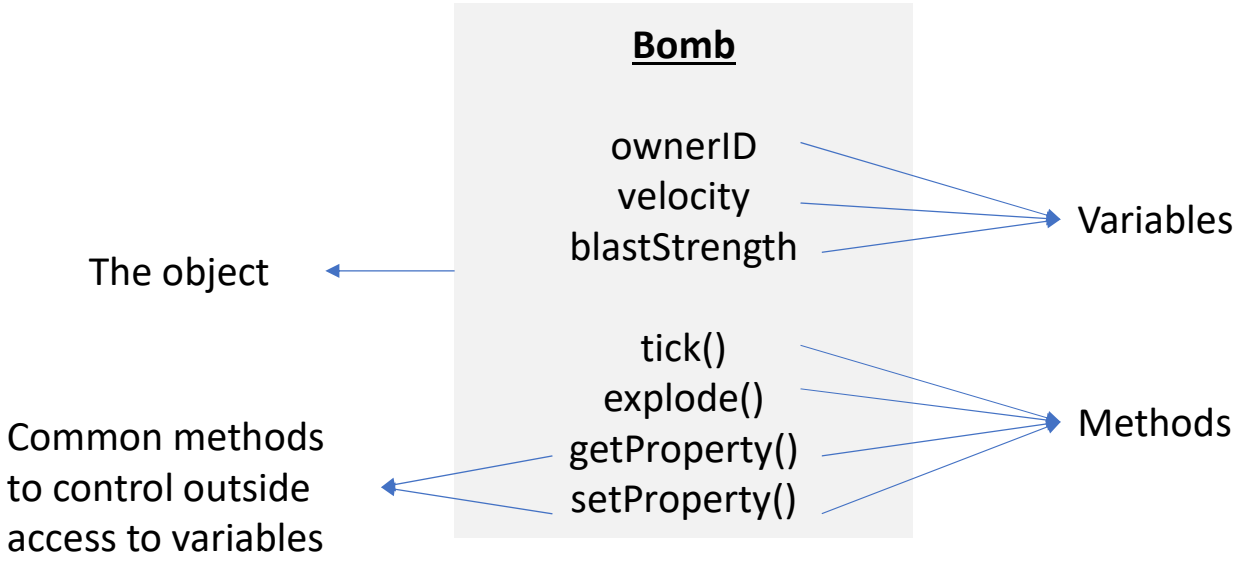
- **Abstraction**: Code grouped into class methods means that other developers need not be concerned with the details of the implementation. This helps with making changes and additions over time.

- **Inheritance**: Objects can be part of a hierarchy of objects with common properties and/or behaviours. This simplifies implementation of objects and reduces development time, as properties and behaviours can be reused while maintaining unique objects with relationships to others.

- **Polymorphism**: Variables, methods and classes can all take on different forms, depending on the context of the execution. For example, there could be two different methods with the same name, but expecting different types of information and performing different logic. Depending on the information passed within the program, the associated logic would be executed. Or, as a second example, the "+" sign in Java can be used as mathematical addition if used with numbers, or for string concatenation if used with text. This often helps reduce code duplication.

OOP - Summary

- Objects are represented as **classes**, entities that exist in the environment we define. They may be part of a hierarchy of objects with common properties and/or functionality.
- Objects have properties, represented as **class variables**. These variables may be other objects.
- Objects can *do* things in the environment defined, and things can be done *to* the objects. These are represented as **methods** (or functions) within the class.



OOP - Summary

Benefits:

- Intuitive, reflective of the real world.
- Useful for large, complex or actively updated and maintained projects.
- Facilitates collaboration.
- Code reusability, scalability and efficiency.

Steps:

1. Identify objects that will exist in the environment / that you wish to manipulate.
2. Identify relationships between the objects.
3. Create classes for each object, with their properties and methods.

The Maze Game – OOP Review

Step 1: Highlight all nouns (green underlined), adjectives/properties/numerals referring to them (bold orange) and verbs (italics blue). Those not important in a sentence can be ignored.

- **6** player *run* around in a **2D** maze, as **2** teams of **3**.
- They can *change* the team they are on at any time by *stepping* onto a button in the **middle** of the maze. *Stepping* on the button *swaps* the player's team, and has an initial **10** frames cooldown, *increasing* by **2** every time a player *uses* the button.
- player can *tag* opponents by *doing a "tag" action* **when next to 1** opponent. Tagged players *are out* of the game and they *lose*. If **2** players *tag* each other in the same game tick, they both *lose*. If player A *tags* player B in the same tick when they themselves *were tagged*, both player A and B *lose*. Tag actions *are invalid* if a player *is adjacent* to **more than 1** opponent (adjacency *does not consider* diagonals).
- The **last** player standing *wins*.
- The game also ends **after 1000 game** ticks. If multiple players from the same team *are alive* at the end, but only **1** player from the **other** team, the **1 solo** player *wins* and everyone else *loses*. If multiple players from both teams *are still alive*, everyone *loses*. If 2 player *are alive at the end* and on opposite teams, they *tie*.

The Maze Game – OOP Review

Step 2: remove all other unimportant words and remove duplicates (or those with similar meaning e.g. change team/swap team here). All nouns in singular form. Can structure things a little for clarity:

- player -> **6** *run* (in) **2D** maze
- player -> **2** team of **3**
- player -> *swap* team
- player -> *step* on button
- button -> (**middle**) maze
- button -> *press* to *swap* player team
- button -> cooldown, *increase* by **2 (when used)**
- player -> *tag* player (opponent) **when next to 1 opponent [+other rules]**
- player -> *lose* **when tagged**
- player -> *win* **if last at the end**
- game -> *end* **after 1000 ticks [+other rules for player win conditions]**

The Maze Game – OOP Review

Step 3: for simplification, remove all orange items. These are important for coding the problems, but not for an abstract representation of the OOP problem. Group bullet points by noun on the left side.

- player -> *run* (in) maze
- player -> team
- player -> *swap* team
- player -> *step* on button
- player -> *tag* player (opponent)
- player -> *lose*
- player -> *win*
- button -> maze
- button -> *press* to *swap* player team
- button -> cooldown, *increase* cooldown
- game -> *end*

The Maze Game – OOP Review

Step 4: Group all elements corresponding to one noun on the left (an object!) under the same bullet points, removing any duplicate properties or logic

- maze
- player
  - *run* (in) maze
  - team
  - *swap* team
  - ~~*step* on button~~
  - *tag* player (opponent)
  - *lose*, *win*
- button
  - (in) maze
  - *press* to *swap* player team
  - cooldown
  - *increase* cooldown
- game
  - *end*

The Maze Game – OOP Review

Step 5: Find any implied properties currently not explicitly stated and add them. Some things might need to be renamed as well.

- maze
- player
  - *run* (in) maze -> *changePosition*
  - position (in) maze ←
  - team
  - *swap* team
  - *tag* player (opponent)
  - *lose*, *win* -> *changeWinStatus*
  - winStatus ←
- button
  - position (in) maze ←
  - *press* to *swap* player team
  - cooldown
  - *increase* cooldown
- game
  - *end*

The Maze Game – OOP Review

Step 6: Group the green items in the sub-bullet points (properties/variables) and the blues (behaviours/methods)

- maze
- player
  - position (in) maze
  - team
  - winStatus
  - *swap* team
  - *tag* player (opponent)
  - *changeWinStatus*
  - *changePosition*
- button
  - position (in) maze
  - cooldown
  - *increase* cooldown
  - *press* to *swap* player team
- game
  - *end*

The Maze Game – OOP Review

Step 7: Rewrite some of the names for clarity. For methods, any green elements left are arguments, or properties modified (can be removed and just point to with an arrow). If other methods are referred to within other methods, they can also be removed and pointed to with an arrow.

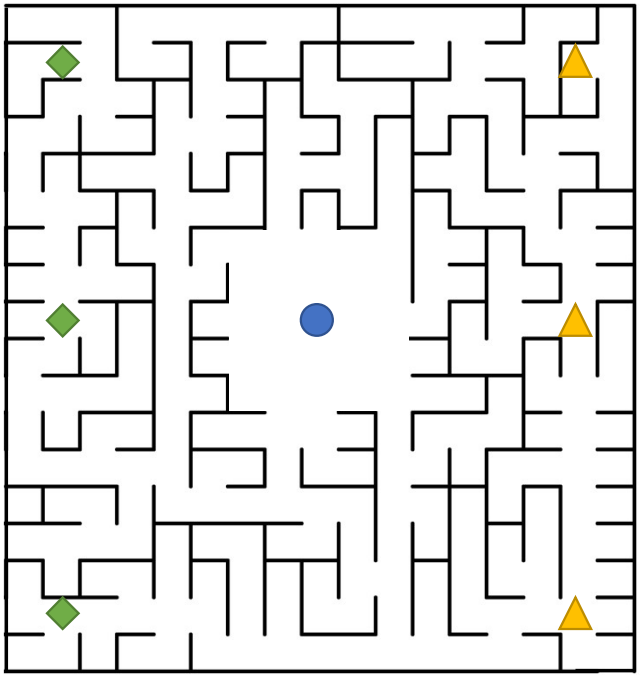
- maze
- player
  - position
  - team
  - winStatus
  - *swapTeam*
  - *tag (player=opponent)*
  - *changeWinStatus*
  - *changePosition*
- button
  - position
  - cooldown
  - *increase*
  - *press*
- game
  - *end*

The Maze Game – OOP Review

That's it! Other interactions may be represented as well (e.g. player pressed button), but these are the key elements for abstracting this problem with objects and their properties and behaviours. For implementation, we'll need to bring back some of the elements removed earlier (in orange), but this gives us an idea of what happens in the program.

- **maze**
- **player**
  - position
  - team
  - winStatus
  - *swapTeam*
  - *tag (player=opponent)*
  - *changeWinStatus*
  - *changePosition*
- **button**
  - position
  - cooldown
  - *increase*
  - *press*
- **game**
  - *end*

[1] Try it out! (10 minutes)



To do:

1. What are the objects in this environment? What are their properties? What functions can they do? What functions can others do *to* them? (answer in this order, to get an overall picture before details)
2. What is the relationship between the objects in the environment?

[1] Try it out! (10 minutes)

1. Find the "Week 2 – OOP.pptx" file, and create a new slide to draw your diagram.

2. Use rectangles to depict classes, write properties or behaviours within the rectangles. Use arrows to draw relationships. PPT was not meant for class diagrams, so a rough representation is fine!

3. It's okay to not model the whole problem, as long as you gain an understanding of how this could be modelled.

4. Note down any aspects that stand out.

Note: If you'd like to be more precise and learn more about class diagrams: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-class-diagram-tutorial/ (in IntelliJ Ultimate, you can right-click on a package -> Diagrams -> Show diagram; to see the class diagram for an existing project)

Possible Game Extensions

- Easy to implement with OOP!
- Small changes in parts of the program to modify/add/remove objects.

- More game objects: traps, pick-ups
- More player actions: jump over walls, dodge tags
- More effects: game button also makes player immune for X ticks
- Game events: randomly assign teams every X ticks
- More object instances: more buttons, different map configurations
- Game variations: change parameters to check variations (e.g. longer delays for buttons)
- ...

Alternative OOP Problem

Think about how you would model the following problem as object-oriented:

- Students in a 12-week university module have 2 assignments to deliver.
- They work in groups of 3 for each assignment, but may be part of different groups for each assignment.
- The level of activity of each student, and each group, is tracked weekly.
- The marks for the assignments are given according to a 5-point marking scheme, with 0 to 20 points awarded for each point in the marking scheme.
- For each assignment, students assign their teammates a contribution mark, from 0 to 10.
- The module organiser wishes to know if there is a correlation between the level of activity of a student, their contributions and their marks. They also wish to know how the overall level of activity changed over time.

1: Java Basics

Variables, types, methods, classes

Java Programs

- **Packages** grouping **classes**, each class with some **methods**
- **Variables** of different **types** store information
- Mathematical operations and control structures modify the information
- The program controls the **data flow**
- **Compiling** a Java program = converting the **source code** (or programmer-readable text in your files) into bytecode (platform-independent instructions for the Java Virtual Machine)
- Always useful, print statements:
  - `System.out.println("some text"); //` Prints to the console
  - IntelliJ shortcut: type "sout" and press the TAB key

[2] Try it out! (5 minutes)

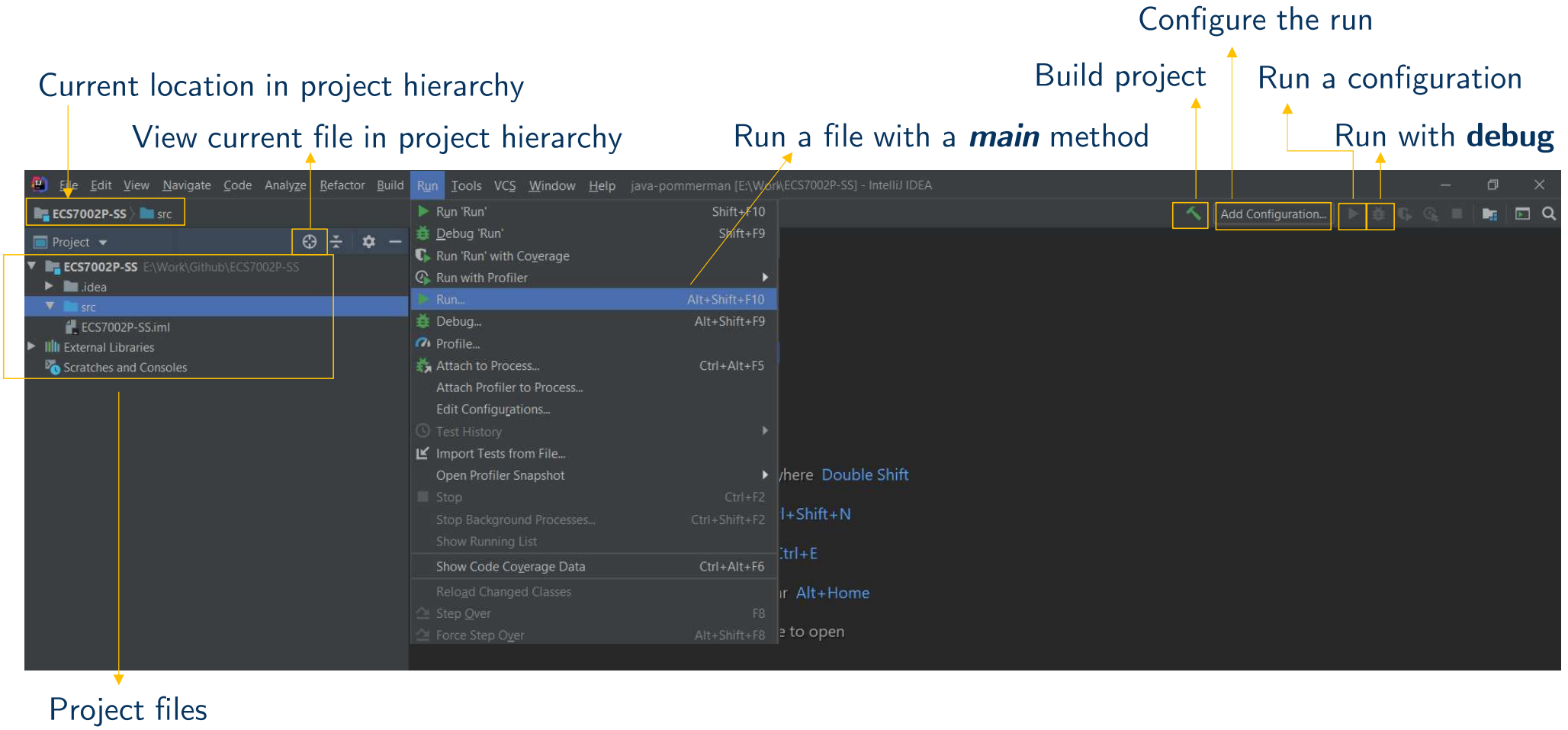
Let's set up an IntelliJ project for these sessions that we'll slowly build up on. IntelliJ is a Java IDE (integrated development environment):

- Source code editing
- Build automation
- Debugging
- Java compiler
- Highlights syntax for clarity
- Highlights syntax errors while writing code
- Makes useful suggestions for completing in-progress code
- Easy navigation through code

To do:

1. Open IntelliJ / download it: https://www.jetbrains.com/idea/download/ (as a student, you can sign up for a student pack to get access to the Ultimate version and other JetBrains software https://www.jetbrains.com/community/education/#students)
2. Create a new project, completely empty (do not use templates, frameworks or anything else, just a basic project).

[2] Try it out! (5 minutes)



Types

- Java is **statically typed** (type is known at compilation time) and **strongly typed** (variable is tied to its type and causes errors if values assigned don't match)
- A **primitive type** is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values.
- Common data types (most common in green):
  - Integer (**int**): 0, 1, -55, 294
  - Long (**long**):
  - Double (**double**): 64-bit real number (3.14, 1.0, -44.3)
  - Float (**float**): 32-bit floating point number (3.14f, 0.0f)
  - Boolean (**boolean**): truth value (true or false)
  - Character (**char**): one character ('c', '%') – always in single quotes
  - [not primitive] String: text ("hello", "world") – always in double quotes

Variables

- Variables are uniquely named locations storing values of a particular type, e.g.
  - `int number;`
- **Assigning** a value (called **initialization** if it's the first value for the variable in the scope):
  - `number = 5;`
- Declaration and assignment can be combined:
  - `int number = 5;`
- Can change type to a compatible other type through **casting**:
  - `float realNumber = (float)number;`
  - This is implicit in some cases, when smaller data types are assigned to larger data types, e.g. int to double

- Can be **local**: the variable only exists within the **scope** (**{}**) it was declared in.
- Can be **global**: declared outside of a method, but within a class, the variable belongs to the class.

- Class variables (global) can be accessed through an **instance** of a class:
  - `object1.number = 3;`

Common Class Variable Modifier Keywords

- Access to the variable can be controlled (e.g. `private int number;`):

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

- **final**: the variable will only have one value assigned on declaration, and its value cannot be changed. (constants)

- **static**: the variable belongs to the class instead of instances of a class. Its value should be retrieved directly from the class, and not an instance of a class:
  - `int number = HelloClass.staticNumber;`

Operations with Variables

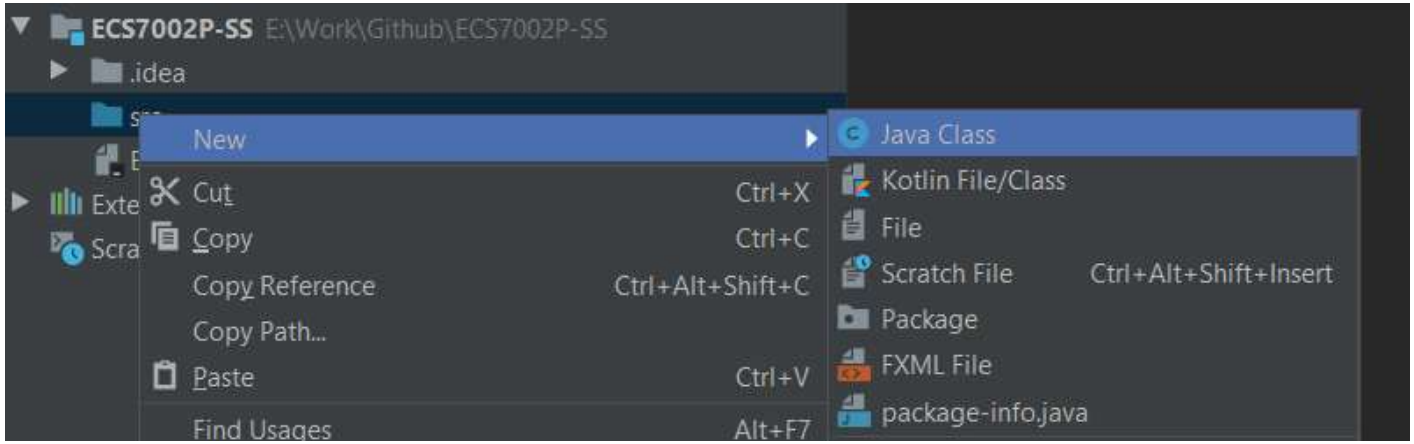
- Numerical types can be used in mathematical operations, e.g. `int otherNumber = number + 2;`
- And can be on both sides of the assignment if they already have a value, e.g. `number = number + 1;` (or in short: `number++;` or `number += 1;`)
  - the second short form works with other operators too: - * / %
- Order of operations like in math, left to right. Parentheses used to increase precedence.

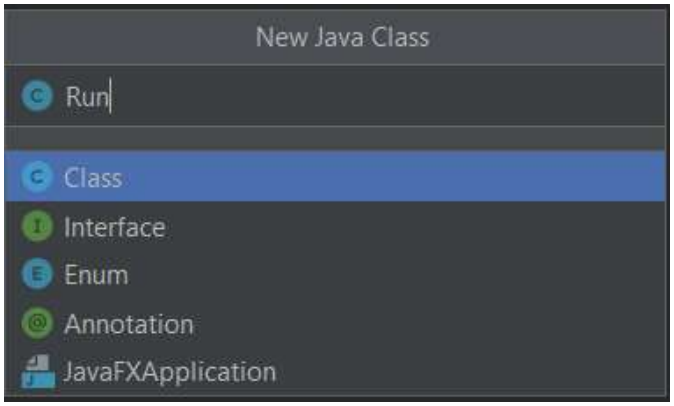
- The String type supports concatenation:
  - `String s = "a" + " b " + 5;  // s = "a b 5"`

- The boolean type supports logical operations: (AND: &&, OR: ||, not equals: !=)
  - `boolean truth1 = false && true;  // false AND true = false`
  - `boolean truth2 = false || true;  // false OR true = true`
  - `boolean truth3 = 3 == 3;  // 3 equals 3 = true`
  - `boolean truth4 = 3 != 3;  // 3 not equals 3 = false`
  - `boolean truth5 = 5 <= 3;  // 5 less than or equal to 3 = false`
  - `boolean truth6 = !truth2;  // not truth2 = not true = false`
  - `truth1 &= truth2;  // truth1 AND truth2 = false AND true = false`

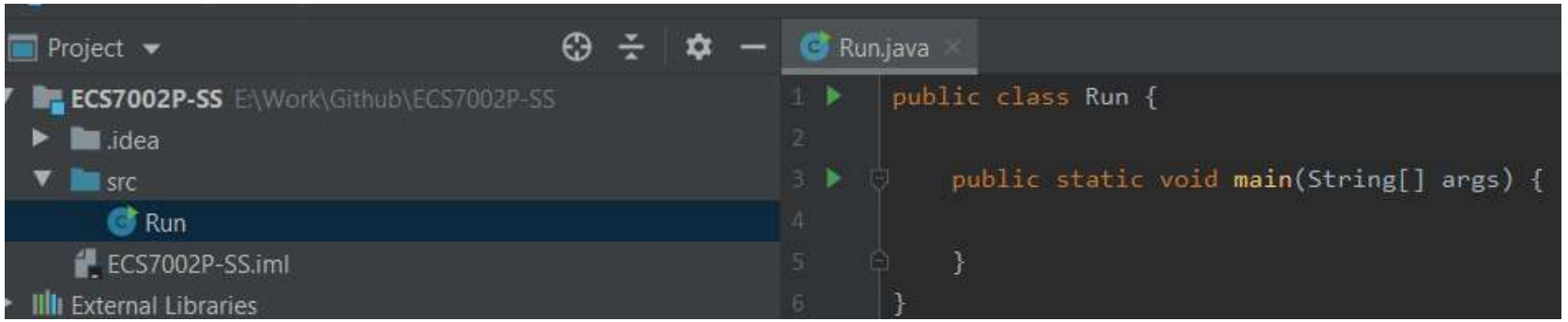
[3] Try it out! (10 minutes)

1. Right click on the "src" folder in the IntelliJ project, and create a new class called "Run"



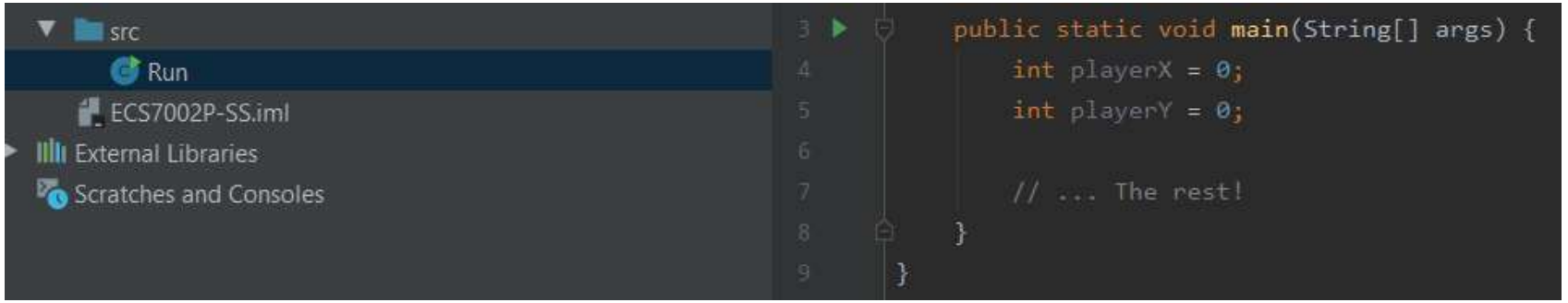


2. Copy the following code (we'll see more about classes and methods soon!), or use the "psvm" + TAB shortcut to create the main method between the curly brackets {} in the new "Run" file.



[3] Try it out! (10 minutes)

3. The code within the curly brackets in the ***main*** method here will run our maze game. However, we'll need more concepts before we can get there. For now, let's consider a very simple scenario:
   a) Only 1 player, with X and Y coordinates in the grid (2 integer variables)
   b) The player has a name (a String variable)
   c) The player will start at top-left position (0,0) and move diagonally towards the bottom-right, once at every game tick.
   d) When the player reaches position (5,5), the game ends.
   e) We keep track of the game tick (integer variable, starting at 0) and whether the game has ended (boolean).
4. Write down how this scenario works between the curly brackets {} of the ***main*** method, printing the following at every game tick: game tick, whether game has ended, player name, player position (X, Y).
5. Remember that command `System.out.println(text);` prints to the console.
6. Only use variables here, nothing more! (you will have to repeat code 5 times, yes).



Control Structures

```
int i = 0;
while (i < 5) {
        statement to repeat 5 times;
        i++;
}
```

```
int i = 0;
do {
        statement to repeat 5 times;
        i++;
} while (i < 5);
```

```
for (int i = 0; i < 5; i++) {
        statement to repeat 5 times;
}
for (ObjectType object: objectList) {}
```

```
if (boolean expression 1) {
        statement;
        continue;
} else if (boolean expression 2) {
        other statement;
} else {
        other statement;
}
```

```
switch (variable) {
        case value1: statement; break;
        default: other statement;
}
```

[4] Try it out! (5 minutes)

1. Rewrite the code from the previous exercise to now use control structures.

2. Add a ***while*** loop which uses the gameEnded boolean variable for its condition, initialised to false.

3. Add an ***if*** statement to check whether the game has ended (X and Y have the correct values) to update the gameEnded variable.

4. What other control structures could be used here instead?

Acknowledgements

- Part of the material inspired by:

  MIT OpenCourseWare http://ocw.mit.edu
  6.092 Introduction to Programming in Java
  January (IAP) 2010