Java for Game AI
2: Java Concepts

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

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

Queen Mary University of London

Outline

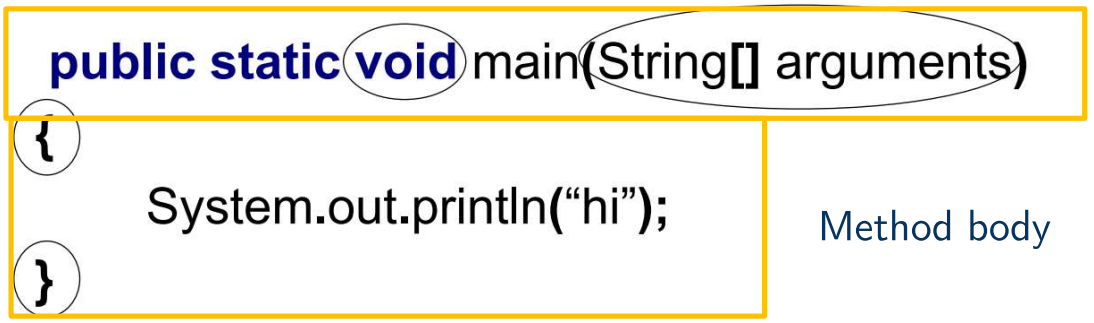
- ✓ Object-Oriented Programming
- ➢ Types, Variables, Methods
- ❑ Classes, Inheritance and Interfaces
- ❑ Java Coding

Methods (1)

- Big programs are built out of small methods
- Methods can be individually developed, tested and reused
- The user of a method does not need to know how it works
- In Computer Science, this is called "abstraction"

- Are made up of a block of code with a specific function, e.g. generate power-up locations in the board in Pommerman
- Identifying the different functions and splitting code up into methods makes code cleaner, more readable and more easily maintainable
  - If something goes wrong, you can isolate the issue within specific methods and fix those without worrying about the rest of the code.

Methods (2)



- The **method signature** is the first line declaring the method:
  - First keyword: access modifier (as before)
  - **`Static`**: method belongs to the class (and not instances of the class) and can only use other **static** variables in the same class, unless a specific instance is being used.
  - Third keyword: **`return`** type
    - Void: does not return any value
    - Variable type: returns a variable of that type
    - Methods only return one variable in Java. If more are needed, use **`collections`** or **`java.util.Pair`** for only 2
  - A method can have 0 or any number of arguments (separated by comma, each with a type specified)
- Curly brackets define the block or **scope** of the method, which contains all method statements (its **body**).

Methods (3)

- Method **signature** = the first line of the method, including keywords, its name and its arguments
- Method **arguments** = variables which are assumed to have some values within the method itself, and are given these values when the method is called.
- **Calling** a method = processing the block of code from the method (with arguments assigned specific values), before returning to the main code

- To call a method, use its name:
  - If in the same class:
    - Void return: `methodName(arguments);`
    - Type return: `ObjectType variable = methodName(arguments);`
  - If in other classes and **not** static, where **variable** is an object of the method's class:
    - Void return: `variable.methodName(arguments);`
    - Type return `ObjectType otherVariable = variable.methodName(arguments);`
  - If in other classes and static:
    - Void return: `VariableClass.methodName(arguments);`
    - Type return: `ObjectType otherVariable = VariableClass.methodName(arguments);`

[5] Try it out! (5 minutes)

1. Write a **`static`** method within our "Run" file (above the ***`main`*** method, but within the brackets {} for the "Run" class).

2. This method should perform one iteration of the simple scenario we worked with before (i.e. moves the player diagonally one step).

3. The method should return true if the game has ended, and false otherwise.

4. Integrate this method within your earlier code in the ***`main`*** method, replacing the parts that are now included in the method instead.

2: Java Concepts

Classes, Inheritance and Interfaces

Classes (1) - Basics

- Defined by the keyword **`class`**.

```
public class Run {
}
```

- The **scope** of a class is defined by its curly brackets {}. Everything within these brackets belongs to the class, and can be called *members* of the class (variables, methods, static variables and static methods).
- Access to classes is controlled by access modifiers, similarly as with variables and methods.
- Variables hold **references** to objects, rather than their **values** (as with primitive types). If the object is modified, all variables holding a reference to that object are also going to hold the same modified version.

```
Bomb bomb1 = new Bomb(10, 5, 2);
Bomb bomb2 = new Bomb(10, 5, 2);
Bomb bomb3 = bomb1;
bomb1.timeToLive --;
boolean truth1 = (bomb1.timeToLive == bomb3.timeToLive);   // True
boolean truth2 = (bomb1 == bomb2);   // False
boolean truth3 = (bomb1 == bomb3);   // True
```

- "==" compares references for objects.
- object1.equals(object2) should be used instead to compare values. This method can be implemented on any custom classes, and exists on the Java built-in types (e.g. String).

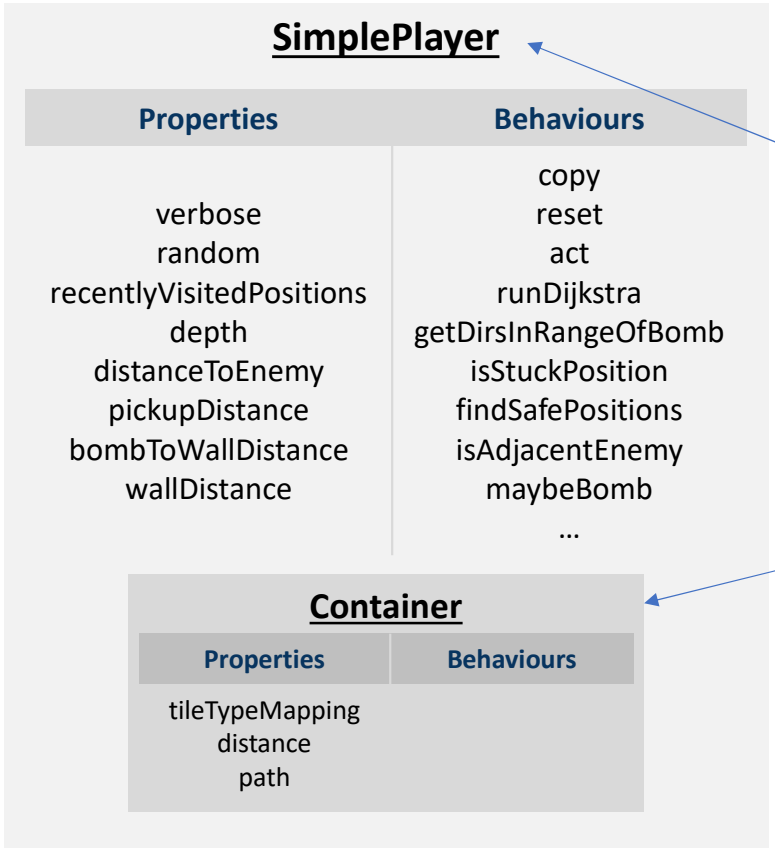
```
@Override
public boolean equals(Object o) {
    return true;
}
```

Classes (2) - Basics

- In Java, generally, 1 object = 1 class = 1 file.

- A `main` method in a class allows you to *run* the class:
  - `public static void main(String[] args) {}`
- All programs should have 1 main method as the entry point, but other classes may have their own main methods as well.
- A class can have **inner (nested) classes**. These are declared within the scope of an **outer** class. Useful for e.g. container classes that group several variables, but are not needed outside of the outer class.
  - Outer classes can only be declared as public or package private.
  - Inner classes can have any access modifier, as long as it is not higher than the outer class.
  - Inner classes can be static.
  - Inner classes can be accessed from outside of the outer class through the outer class.

Classes (2) – Inner Classes



Outer class

Inner class

In code:

```
public class SimplePlayer {

        …

    public static class
    Container {
        HashMap tileTypeMapping;
        HashMap distance;
        HashMap path;
    }

        …
}
```

Classes (3) - Constructors

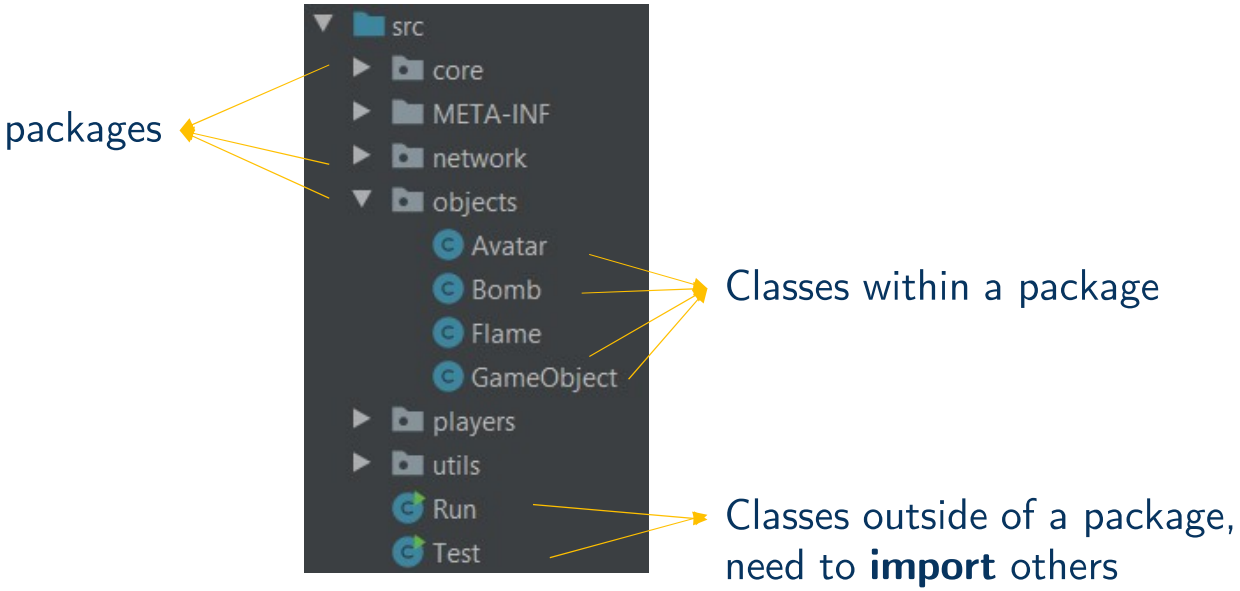
- To create an object represented by a particular class (or **instantiate** an object, or create an **instance** of a class), the keyword **new** is used:
  - `Run runObject = new Run();`
- This statement is an implicit call to the **constructor** of the object (a special type of method).
- Constructors have no return type, and their name is always the class name.
- The default constructor:
  - `public Run() {}`
- A class can have multiple constructors, with different types of arguments (the correct one would be called based on the arguments given when the object is created):

```
public Run(int nTimes) {
    Run();  // We can call other constructors from here as well
    this.nTimes = nTimes;
}
…
Run runObject = new Run(5);
```

- Keyword **this** refers to the object instance we are creating. It can also be used in other methods to refer to the object members (and avoid confusion with the same variable/method names).

Classes (4) - Packages

- Classes can be grouped into packages (= directories), and packages can be nested, for a clean logical structure, or to separate files with similar names. The **`package`** is the first thing declared in a file.
- Classes declared as **`public`** can be accessed by others outside the package through an **`import`** statement, e.g. `import utils.Vector2d;` This can be handled automatically in IntelliJ by typing out the statement using the external class, and using the shortcut ALT + ENTER.
- Classes within the same package have access to all other classes in the same package and do not need to import them.



[6] Try it out! (10 minutes)

Remember the earlier exercise game modelled as object-oriented:

- 6 players 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 players uses the button.
- Players 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 players 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 players are alive at the end and on opposite teams, they tie.

[6] Try it out! (10 minutes)

To do:

1. Create a *core* package in the project.
2. Within this package, create an abstract class **`Player`** that will hold: player ID (integer), random seed (long), a random number generator (`java.util.Random` object), team ID (integer, 0 or 1), game status (integer, -2 = undecided, -1 = lose, 0 = tie, 1 = win), number of opponents tagged (int), position (integer)
3. The `Player` constructor receives and sets 1 argument, the random seed. It also sets game status to -2, and #opponents tagged to 0. Finally, it initialises the random generator to a new `Random` object, given seed.
4. The `Player` class has the following methods (implement their contents!):
   - `void setTagged(Player other) {…}`
     - If this method is called, then the player was tagged and they lose. Set their game status and increase the other player's count of opponents tagged.
   - `void setGameStatus(int newStatus) {…}`
     - This method should set the player's game status to the argument received.
   - `void swapTeam() {…}`
     - This method should swap the player's team (If 0, the team becomes 1. If 1, the team becomes 0)
   - `protected abstract int act();`
     - This method will be called when it is this player's time to return an action. We consider actions to be integers, 0 – do nothing, 1 – move up, 2 – move right, 3 – move down, 4 – move left, 5 – tag. This method is abstract.

Classes (5) – .jar Files and External Libraries

- In Java, any program can be compiled to a .jar file which can then be run via command line as:
  - `java –jar file.jar [arguments]`
- To build a .jar file in IntelliJ, a 2-step process to set-up:
  1. File -> Project Structure -> Artifacts -> "+" -> JAR -> From modules with dependencies ... -> select main class (entry point to the program, with a ***main*** method) -> Apply -> OK
     - This creates a new folder in src/ called "META-INF", which contains only one "MANIFEST.MF" file with basic information about the program compiled.
     - Note that there can be only 1 such file in the project – if this already exists, you can reuse it when creating the artifact, or simply delete it before following this step.
  2. Build -> Build Artifacts -> select the correct artifact by name, in case multiple -> Build
     - This creates the actual .jar file, found in: project_root/out/artifacts/artifact_name_jar/artefact_name.jar
- This file can then be run from the command line, or imported into another project as a **library** to give access to their public classes (and all of their public members).
- To import a library in IntelliJ, 2 ways:
  1. Copy the .jar file in a "project_root/lib/" directory in your project. Right-click the file -> Add as library
  2. File -> Project Structure -> Libraries -> "+" -> Java -> Find .jar files -> OK -> Apply -> OK
- Classes from an imported library can be used with **`import`** statements as if in a different package.

Inheritance (1)

- Classes can **`extend`** from other classes. Classes that extend are called sub-classes, while extended classes are called super-classes. Sub-classes can access (*inherits*) any public or protected members of the super-class. Explicit access is done through the keyword **`super`**.
- Sub-classes **MUST** provide a constructor that matches the super-class's constructor, if the super-class has any constructor explicitly implemented. The call to the super constructor needs to be the first thing called in the sub-class: ***`super(arguments)`***.
- Sub-classes can **`override`** methods from the super-class, by creating methods with the same **signature** as in the super class, but different content.

- All classes implicitly extend from the Java super-class **`Object`** and can override its methods, such as **`equals`**, **`hashCode`**, **`toString`**.

- **Limitation:** A class can only extend **one** other class.

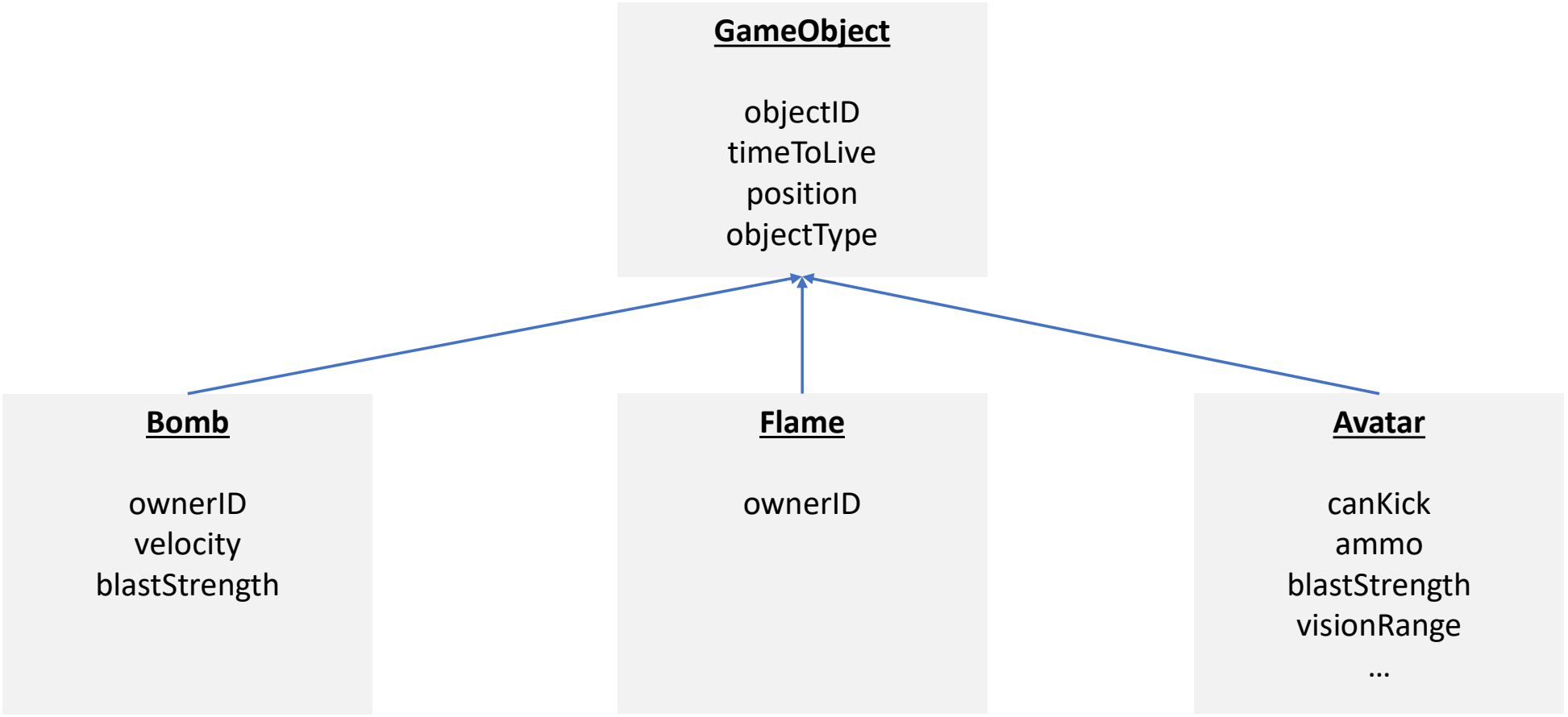
Inheritance (2)

- Sub-classes **MUST** provide an implementation of any methods declared as `abstract` in the super-class.
  - Abstract methods declared in a class only specify the signature of the method, but not its contents.
- If a class contains any abstract methods, the class itself must be declared **abstract**.
- Abstract classes can **not** be instantiated. A sub-class must be created (non-abstract) which can then be instantiated itself. The type of the variable holding the instance **can** be the abstract super-class.

- If class X extends class Y, an object of type X is also a type Y (e.g. a bomb is also a game object), but not the other way around. Operator `instanceof` can be used in a boolean expression to check the type (class) of an object (in other words, if an object belongs to a class), for example:
  - `bomb1 instanceof Bomb // true`
  - `bomb1 instanceof GameObject // true`
  - `bomb1 instanceof Avatar // false`

Inheritance (3)

- Remember how Bomb, Flame and Avatar are all different types of game objects with common properties:



Inheritance (4)

In code:

```
public class GameObject {

        int objectID;
        int timeToLive;
        Vector2D position;
        TileType objectType;

        public GameObject(TileType type, int x, int y) {
                this.objectType = type;
                this.position = new Vector2D(x, y);
                this.objectID = 0;
                this.timeToLive = 10;
        }

        …
}
```

```
public class Bomb extends GameObject {

        int blastStrength;
        int playerId;
        Vector2D velocity;

        public Bomb(int x, int y, int b, int p, Vector2D v) {
                super(BOMB, x, y);
                this.blastStrength = b;
                this.playerId = p;
                this.velocity = v;
        }

        …
}
```

Inheritance (5)

In code:

```
public abstract class Player {

    protected int playerID;
    protected long randomSeed;

    protected Player(long seed, int pID) {
        ...
    }

    public abstract ACTION act(GameState gs);

    ...
}
```

```
public class SimplePlayer extends Player {

    ...

    public SimplePlayer(long seed, int pID) {
        super(seed, pID);
        ...
    }

    @Override
    public ACTION act(GameState gs) {
        ...
        return ACTION_NULL;
    }

    ...
}
```

[7] Try it out! (10 minutes)

1. Continuing from the previous exercise, create a *players* package, and a **RandomPlayer** class within the *players* package.
2. This class should extend from the *Player* class and should have a matching constructor which calls the *super* constructor.
3. This class will implement the abstract *act()* method and return a random integer between 0 and 5 (hint: check the `Random.nextInt(int bound)` method).

4. In the *core* package, create a **Button** class. This will hold: position (integer), cooldown (integer), cooldownCounter (integer), currentlyActive (boolean). Additionally, it will implement the following methods:
   - `void tick() {…}`
     - This method will be called once every game tick. If the button is not currently active, the cooldownCounter is reduced until it reaches 0, at which point the button becomes active (and the cooldownCounter is reset to the cooldown value).
   - `void press(Player who) {…}`
     - This method will be called if a player steps on the button when it is active. It will call the player's swapTeam() method, set the button to inactive, increase the cooldown value by 2, and reset the cooldownCounter to the cooldown value.
   - `Constructor:`
     - Receives the button's position, sets currentlyActive to false, the cooldown and cooldownCounter to 10.

Interfaces

- Methods are the way an object interacts with its environment. These form the *interface* between the object and the environment.
- In Java, **interfaces** group together method signatures (without any content) to define how an object might interact with its environment. Specific implementations depend on the object that **`implements`** the interface. A class can implement **any number of interfaces**, separated by a comma.
- Interfaces can implement default behaviours for the methods, by using the keyword **default**.
- If a method does not have a default implementation, it **MUST** be implemented in classes implementing the interface. Otherwise, it can be overridden if needed. Similar to abstract classes, but only with methods.
- Useful for objects with similar behaviours, but different properties otherwise.

```
public interface ParameterSet {

    void setParameterValue(String param, Object value);
    Object getParameterValue(String root);
    ArrayList<String> getParameters();
    Map<String, Object[]> getParameterValues();
    Pair<String, ArrayList<Object>> getParameterParent(String parameter);
    Map<Object, ArrayList<String>> getParameterChildren(String root);
    Map<String, String[]> constantNames();

    default void translate(int[] values, boolean topLevel) {
        …
    }
}
```

```
public class RHEAParams implements ParameterSet {

    …

    public void setParameterValue(String param, Object value) {
        …
    }
}
```

2: Java Concepts

Java Coding

Some Java/IntelliJ Tips & Tricks

- Useful Built-In Classes:
  - `java.lang.Math`
    - **`Math.sin(x)`**
    - **`Math.cos(x)`**
    - **`Math.pow(x, y)`**
    - **`Math.log(x)`**
    - **`Math.PI`**
    - ...
  - `java.util.Pair`
    - **`Pair<Type1, Type2> p = new Pair(object1, object2);`**
  - `java.util.Random` (`nextInt()`, `nextDouble()` etc.)
  - https://docs.oracle.com/javase/7/docs/api/
- IntelliJ common shortcuts:
  - **Comments**: CTRL + / or CMD + /
  - **Jump to method definition**: CTRL + B or CMD + B
  - **Suggest fixes to errors**: ALT + ENTER
  - **System.out.println**: write "sout" and press TAB
  - **Public static void main method signature**: write "psvm" and press TAB

Best Practices (1)

1. Use meaningful names for packages, classes, methods, variables. Conventions:
   a) Packages: nouns, lower case (e.g. players)
   b) Classes: nouns, mixed case with first word capitalized (e.g. HelloWorld)
   c) Interfaces: like classes, but preceded by "I" (e.g. IHelloWorld)
   d) Methods: verbs, mixed case with first word lower case (e.g. sayHi)
   e) Variables: mixed case with first word lower case (e.g. floatNumber)
   f) Constants: Upper case (e.g. PI), separated by underscore (e.g. "MAX_WIDTH")

2. Use indentation (1 tab/ 4 spaces recommended), with each statement on a new line.

3. Use white spaces next to mathematical operators.

4. 1 blank line that separates blocks of code helps readability.

5. Avoid String values for variables that will be checked multiple times: use constants instead to avoid spelling errors.

Best Practices (2)

5. Comment your code:
   a) // for single comment (CTRL + / or CMD + / shortcut in IntelliJ)
   b) /* for block comment, may contain multiple lines */
   c) /***
      * for method comments, to specify what it does, what arguments it receives and what it returns
      */

6. Use "TODO" or "FIXME" keywords in comments, highlighted syntax for easy tracking.

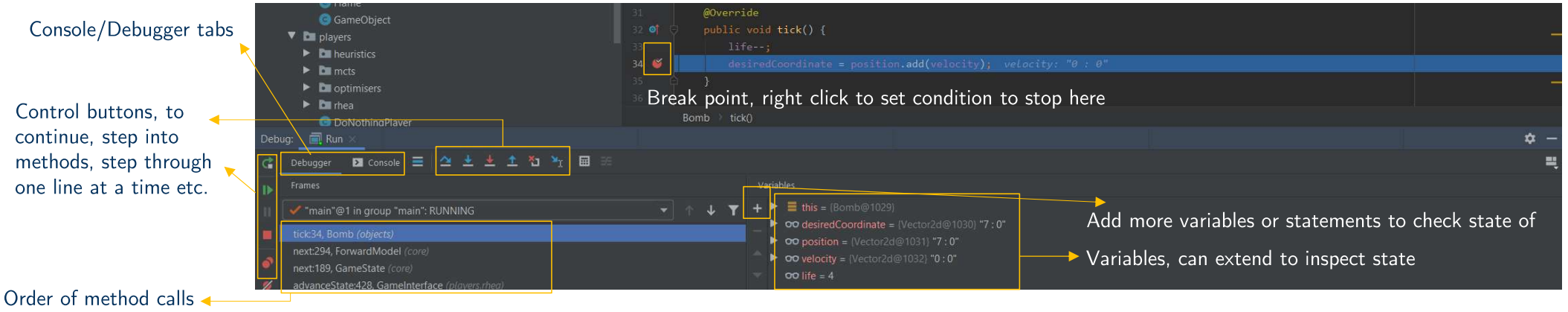
7. Use curly-brackets for all control structures in a consistent style.

8. Split code up into methods, classes, packages. Set the lowest access level necessary for the program to work, avoiding public if possible. Variables should be private as much as possible, with getter/setter methods controlling the access.

9. Avoid hard-coded values in the middle of code, use variables where necessary.

Debugging

- The process of finding and correcting errors in a program.
- In Java:
  - Use print statements to track the values of variables before and after they are modified, to make sure the intended behaviour is actually executed. Or, to track the order of method calls (e.g. super/sub classes). Or, to track control structures behaviour (e.g. is the correct branch of an **if** statement accessed).
  - Use *if (condition) { print statement or other processing }* dummy statements to check specific conditions.
  - https://dzone.com/articles/50-common-java-errors-and-how-to-avoid-them-part-1
  - Other online tools, e.g. http://www.pythontutor.com/java.html
- IntelliJ debugger key aspects:
  - Click next to the line number to create a **break point**. When run in debug mode (Run -> Debug...), the program execution will stop at the break point and allow you to inspect the current state of the environment at that point.



Debugging (2)

- Testing software: https://www.vogella.com/tutorials/JUnit/article.html
- More on IntelliJ Debugger tool: https://www.jetbrains.com/help/idea/debugging-code.html

General steps for debugging:

1. Don't make mistakes: reuse code, design your code first, follow best practices.
2. Find mistakes early: test code regularly; IntelliJ warnings, syntax corrections, compilation errors.
3. Reproduce the error: how can you repeat the error? Design a simple test scenario. Roll back to a previously working version (**version control is important!**) and add changes back in one at a time to see where it breaks. Remove things that are not needed.
4. Generate hypothesis: what could be going wrong?
5. Collect information: if X is your problem, how can you check it? -> print statements, debugger tools
6. Examine data: if it turns out X was indeed the problem, fix it – otherwise, back to step 4.

[8] Try it out!

1. Revise the code written so far according to the list of best practices.

2. Try to create simple test scenarios (create `RandomPlayer` and `Button` objects in the ***`main`*** method in the **`Run`** file and call their methods).

3. Use simple print statements (or the IntelliJ debugger) to make sure your implementation of the methods in both classes does what it's supposed to.

Acknowledgements

- Part of the material inspired by:

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