

This lab has **six (6)** checkpoints.

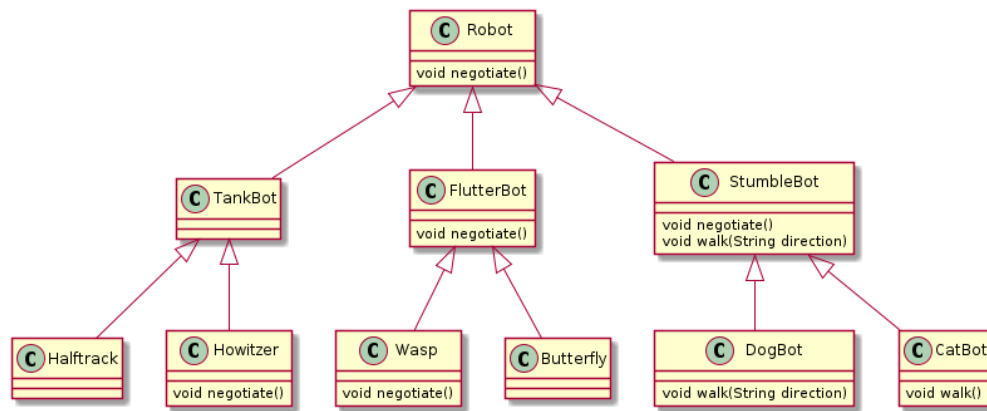
Learning Outcomes

Upon completing this lab, students should be able to

- Define a Base application class with a *protected* data field.
- extend the base class to create two *different* applications.
- Be able to use Gradle commandline property definition to run a specific Java class.

Introduction

1. Dr. Ladd implemented the following class hierarchy. The arrows all refer to an *extends* relationship.



(a) Trace which functions are called in order in the loop.

```
List<Robot> bots = new ArrayList<Robot>();

bots.add(new Robot());
bots.add(new TankBot());
bots.add(new FlutterBot());
bots.add(new StumbleBot());
bots.add(new Halftrack());
bots.add(new Howitzer());
bots.add(new Wasp());
bots.add(new Butterfly());
bots.add(new DogBot());
bots.add(new CatBot());

for (Robot bot : bots) {
    bot.negotiate();
}
```

- (b) As in the previous question: list the specific functions that are executed in order in the loop. (But this is the *same* question as part *a* above. Is it? Really?)

```
List<StumbleRobot> walkies = new ArrayList<StumbleRobot>();

walkies.add(new StumbleBot());
walkies.add(new DogBot());
walkies.add(new CatBot());

for (StumbleRobot walker : walkies) {
    walker.walk();
}
```

✓ Show and explain your answers to these questions to the lab instructor.

2. What is the content of basket after this code executes? You need to indicate the contents and the index of each. You may want to use Java documentation (<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>) to help figure it out.

```
import java.util.Arrays;
...
String fruit[] = { "apple", "pineapple", "pomegranate", "mango", "peach" };

List<String> basket = new ArrayList(Arrays.asList(fruit));
basket.remove("pineapple");
basket.addAll(1, List.of("bacon", "egg"));
basket.remove(4);

/* Thought: could you perform these operations if basket
   were an array of String? Exam questions loom. */
```

✓ Show and explain your answer to the lab instructor.

3. Initialize a Gradle application project in a new directory. Also initialize git in the root of the project.

The name of the project: framework. Starting package: application.

Change the name of application.App to application.Base. Remember that this requires changing the name of the class, the name of the file, and the name of the mainClass in build.gradle.

Delete everything in src/test/java so that your newly renamed application can be built with gradle build.

Add the run block to build.gradle that connects gradle's standard input to that of your program when using gradle run.

Confirm that you can build your application with gradle build.

Confirm that you can run your program with gradle run -q --console=plain.

Commit your working initial version to the git database.

✓ Show the lab instructor

- That `gradle clean` runs (and deletes all compiled artifacts).
- That you can `gradle build`.
- That you can `gradle run -q --console=plain`.
- That `git status` shows a clean working tree.

Be prepared to show the contents of `build.gradle` and/or `Base.java`.

4. Modify `Base` to use the standard construct/run pattern we use in this class: `main` constructs a `Base` object and then calls `run` on the new object.

You should have just two functions in your program now: `main` and `run`.

Add a constructor for `Base` that takes an array of `String`:

```
public Base(String [] args) ...
```

Add a private field to `Base`, a `List` of `String` called `arguments`. In the constructor, use code like that in question 2 above to set `arguments` to refer to an `ArrayList` containing the arguments.

Add a fourth function (after the constructor, `run`, and `main`), `processArgument`:

```
public void processArgument(String argument) ...
```

In `Base`, `processArgument` should print a star, ignoring its parameter's value.

Modify `run` to call `processArgument` with each value in `arguments`. Then, after the loop, start a new output line.

This method, `run`, will be unchanged in the child classes of `Base`; `processArgument` will be changed so that, polymorphically, each program will do something different with each argument.

Use the `--args` parameter with `gradle run` to test out your code:

```
$ gradle run -q --console=plain
$ gradle run -q --console=plain --args duck
*
$ gradle run -q --console=plain --args "duck duck goose"
***
```

Commit your working changes to `git`.

✓ Show the lab instructor your changed files, run the code for them, and show them the clean working tree.

5. Create a **new** package, `io`. Create a new class, `io.Echo` that extends `application.Base`.

`Echo.processArgument` **echos** the argument it is passed to standard output. `Echo` will have three functions:

- `Echo(String [] args)` This non-default constructor passes its argument to the `Base` constructor.
- `processArgument(String argument)` Print the value of the argument followed by a single space.
- `main(String [] args)` Constructs a `Echo` object and calls `run`.

In order to run Base or Echo with `gradle run`, you need to modify the application portion of `build.gradle`:

```
application {  
    mainClassName =  
        project.getProperties().getOrDefault("alternateMainClass", 'application.Base')  
}
```

Meaning: If `alternateMainClass` is defined, then use its value for the `mainClassName`; if it is *not* defined, use `'application.Base'` as the value for `mainClassName`.

If you repeat your previous `gradle run` commands, they should work as before. To see the echoed values (to run `io.Echo`), define the *property* `alternateMainClass` on the commandline:

```
$ gradle run -q --console=plain -PalternateMainClass='io.Echo'  
$ gradle run -q --console=plain -PalternateMainClass='io.Echo' --args duck  
duck  
$ gradle run -q --console=plain -PalternateMainClass='io.Echo' --args "duck duck goose"  
duck duck goose
```

Commit your working changes to git.

✓ Show the lab instructor your changed files, run the code for them, and show them the clean working tree.

6. One more `io` application to extend Base: Call it `io.Cat`.

`Cat.processArgument` opens the file named in the argument, reads it by *line*, and copies (concatenates) the contents to the screen. Something along the lines of:

```
Scanner fin = new Scanner(...around opened file);  
while (fin.hasNextLine()) {  
    line = fin.nextLine();  
    println(line);  
}
```

Commit your working changes to git.

✓ Show the lab instructor your changed files, run the code for them, and show them the clean working tree.

7. [Not a separate checkpoint but necessary] Add and commit your full `gradle` project (if you `git add` just the root directory, `.,` git will add all subdirectories recursively).

Create a repository in your Gitea turn-in organization, associate it with your lab work, and push the repo. This will be done at the end of every lab.