

CS 398: Application Development

Development Projects

IntelliJ IDEA; Build systems; Gradle

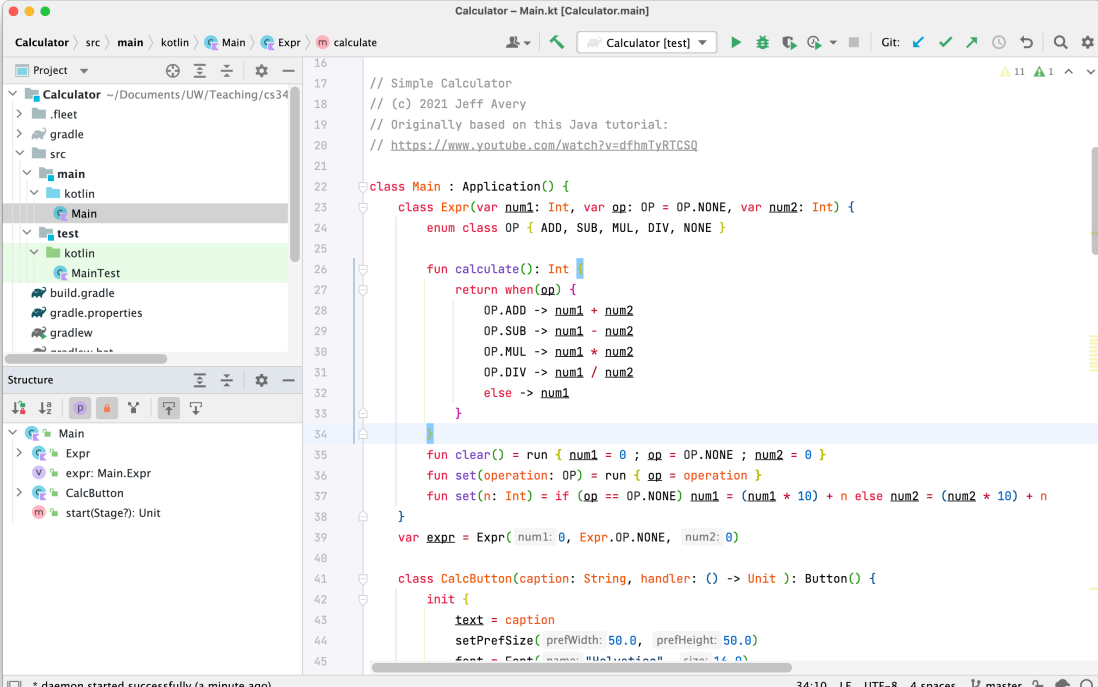
IntelliJ IDEA

Setup your Development Environment

We're going to use **IntelliJ**, an IDE which provides all development functionality, and integrates with all of our tools.

It can be downloaded from <https://www.jetbrains.com/idea/>.

There is an Open Source Community version, which is fine for this course. It runs equally well on Windows/Linux/Mac.



```
16
17 // Simple Calculator
18 // (c) 2021 Jeff Avery
19 // Originally based on this Java tutorial:
20 // https://www.youtube.com/watch?v=dFhmTYRtCSQ
21
22 class Main : Application() {
23     class Expr(var num1: Int, var op: OP = OP.NONE, var num2: Int) {
24         enum class OP { ADD, SUB, MUL, DIV, NONE }
25
26         fun calculate(): Int {
27             return when(op) {
28                 OP.ADD -> num1 + num2
29                 OP.SUB -> num1 - num2
30                 OP.MUL -> num1 * num2
31                 OP.DIV -> num1 / num2
32                 else -> num1
33             }
34         }
35
36         fun clear() = run { num1 = 0 ; op = OP.NONE ; num2 = 0 }
37         fun set(operation: OP) = run { op = operation }
38         fun set(n: Int) = if (op == OP.NONE) num1 = (num1 * 10) + n else num2 = (num2 * 10) + n
39     }
40
41     var expr = Expr(num1: 0, Expr.OP.NONE, num2: 0)
42
43     class CalcButton(caption: String, handler: () -> Unit): Button() {
44         init {
45             text = caption
46             setPreferredSize( prefWidth: 50.0, prefHeight: 50.0)
47             font = Font( name: "Helvetica", size: 14.0)
```

Creating a new project

Choose the type of project

From the splash screen, select “New Project”.

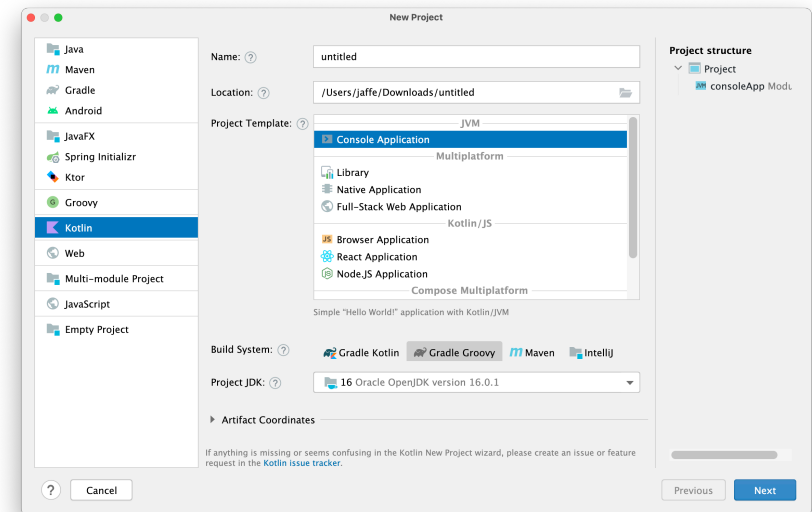
For this course, we recommend one of these options:

Console

- **JVM - Console Application:** Kotlin/JVM for an application that requires the JVM to run.
- **Multiplatform - Native Application:** Kotlin/Native for a standalone application under a specific platform.

Graphical Desktop

- **Java FX - Java FX Desktop Application:** imperative application targeting the desktop JVM.
- **Compose Multiplatform - Compose Desktop Application:** declarative application targeting the desktop JVM.



<https://www.jetbrains.com/help/idea/get-started-with-kotlin.html>

One of the strengths of this ecosystem is that we can use it for practically *anything*.

Creating a new project

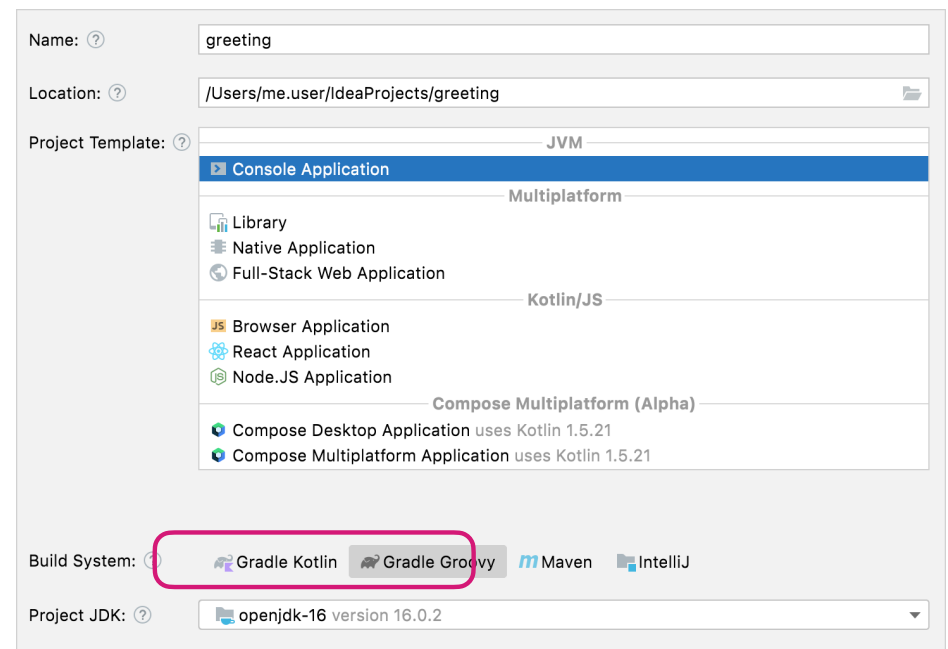
Specify Project Parameters

Make sure to pick **Gradle** as your Build System*.

- Your choice of Kotlin or Groovy determined which programming language will be used in the config files.
- Either is fine, though course examples are mostly in Groovy (they were created before Kotlin was widely supported).

Pick the correct version of **Java JDK**.

- IntelliJ will prompt you to install one if you don't have it installed already.



<https://www.jetbrains.com/help/idea/get-started-with-kotlin.html>

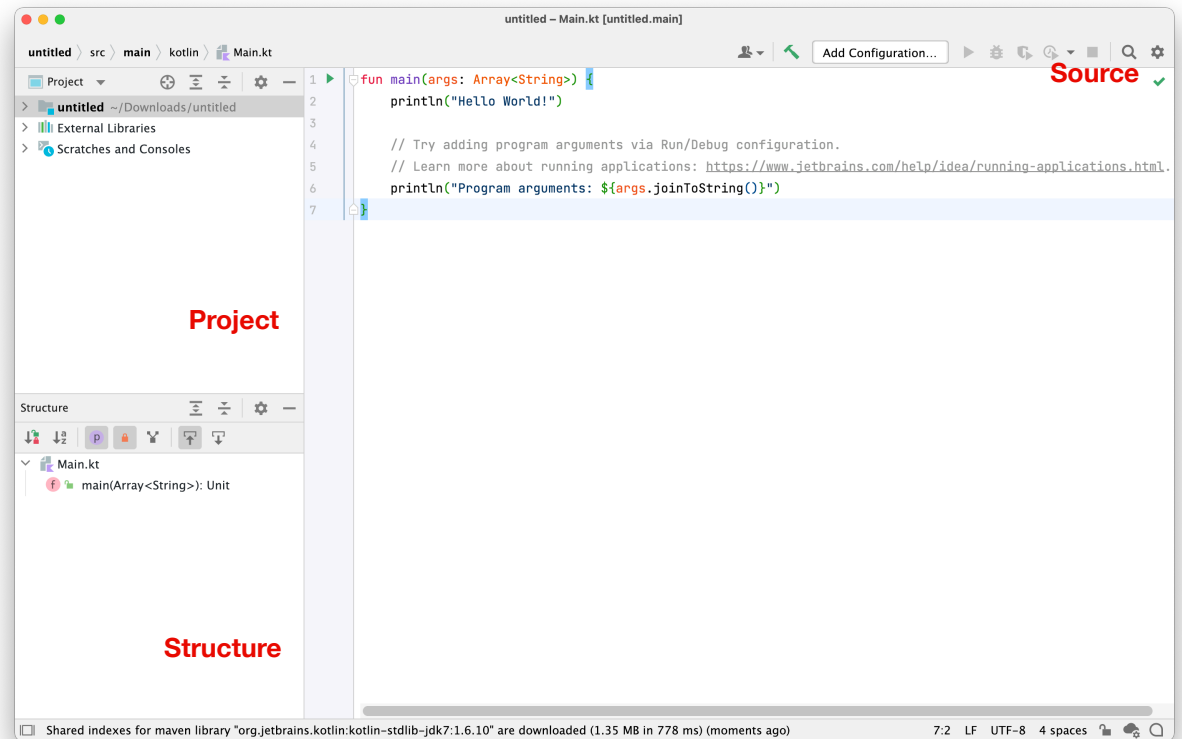
* we'll talk more about Gradle and build systems soon.

Navigating IntelliJ

Layout

IntelliJ has a number of windows:

- **Project:** a list of all files (Cmd-1*)
- **Structure:** methods and properties of the current open class/source file (Cmd-7).
- **Source:** the current source files (no hotkey).
- **Git:** Git status and log (Cmd-9) - not shown.
- **Gradle:** tasks that are available to run (no hotkey) - not shown.



* All keyboard shortcuts are customizable in Preferences. For example, the Gradle window doesn't have a shortcut, but I usually assign it to Cmd-0. Also, Windows/Linux users, lacking a Cmd key, should substitute Ctrl for Cmd.

Navigating IntelliJ

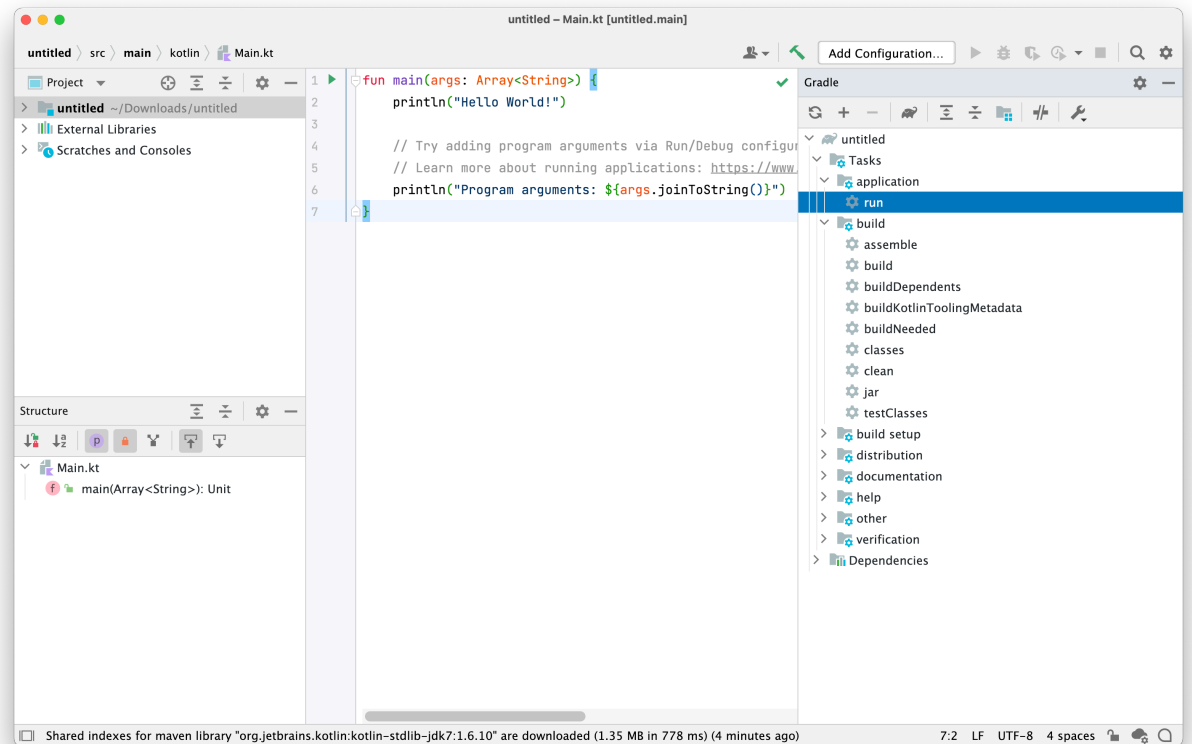
Running your project

View - Tool Windows - **Gradle**

Expand **Tasks**

- Application
 - Run — execute your program
- Build
 - Build — rebuild it
 - Clean — remove temp files

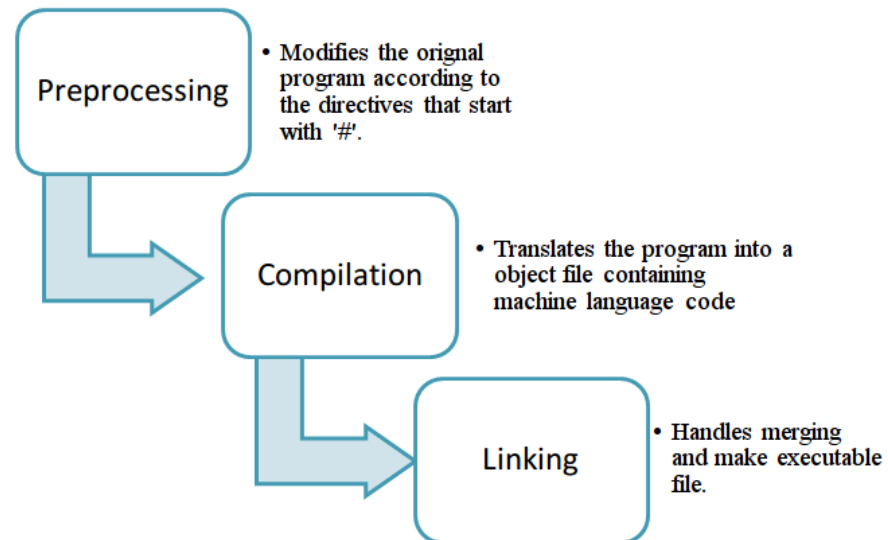
After you build/run once, the toolbar Run and Debug items will be active.



Build Systems

Build Systems

Let's talk about compiling source code. Compilation takes source files from different locations, compiles and links the output to create an executable.



C Compilation Process

However, this **process** often requires more than just compiling and linking code. Before you compile anything, you might need to:

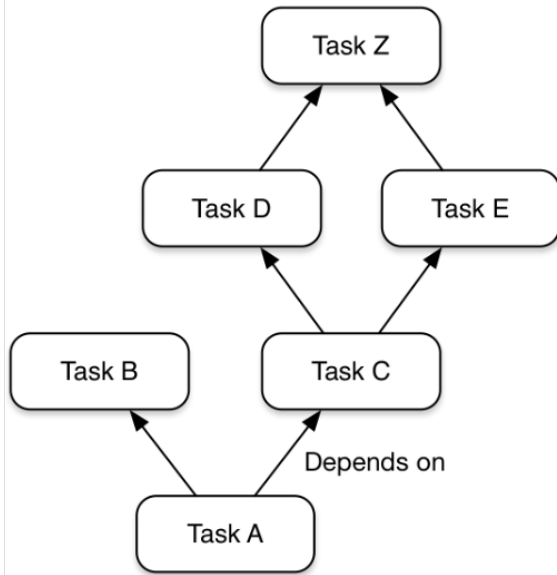
- Download/import new versions of libraries (dependencies).
- Copy resources (graphics files, sound clips, preference files) into a directory structure.
- Run a code analysis tool against your source code to check for suspicious code, formatting etc. or run a documentation tool to generate revised documentation.

After compiling, you will want to:

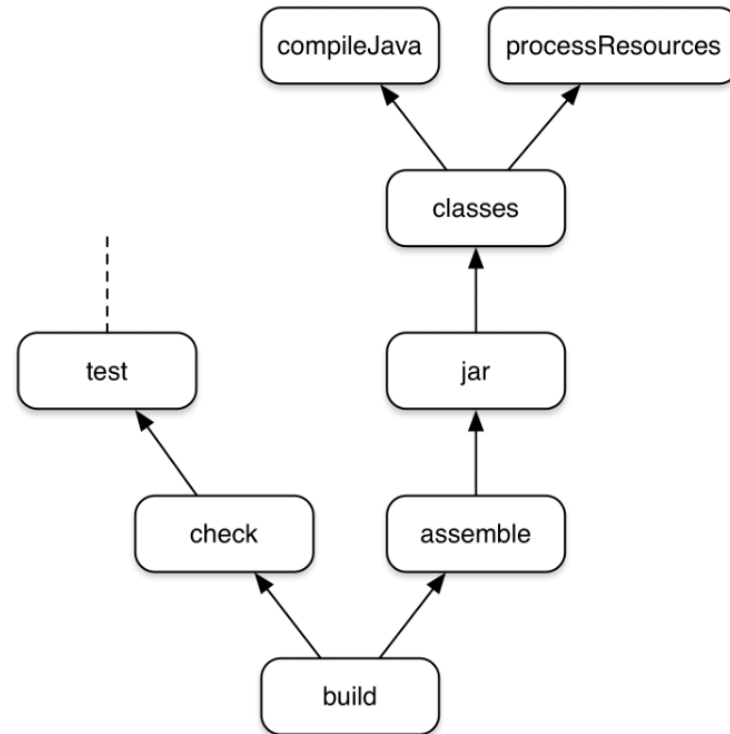
- Test your code to make sure it works properly (e.g. multiple environments).
- Create an installer that you can use to deploy everything.

*Compiling refers to just compiling and linking. **Building** refers to this complete set of steps.*

Generic task graph



Partial task graph for a standard Java build



Task graph for a typical Java build

Performing these steps manually is error prone, and very time-consuming.

The ideal system would be automated and have the following properties:

1. The system would guarantee **consistency** in my builds.
2. It would be **expressive** enough to let me script any task that I need to perform.
3. It would **integrate with other systems** so that I could report results, or delegate responsibility (e.g. to remote test under a different OS).
4. Tasks could be initiated in **response to external events**.
 - e.g. import the newest version of a library when it's published.
 - e.g. rebuild and test when someone commits to the Test branch.
5. Tasks could be **scheduled**.
 - e.g. rebuild and test everything nightly at 2 AM, and email the manager with the git blame results of the person that broke the build.

Systems that do these things are called **build systems** — software that is used to build other software.

Build systems provide consistency in how software is built, and let you automate steps that are required to build, test and deploy software.

They addresses issues like:

- How do I make sure that all of my steps (above) are being handled properly?
- How do I ensure that everyone is building software the same way i.e. compiling with the same options?
- How do I ensure that tests are being run everytime someone builds?

GNU Make

Make is widely used to script builds (by creating a makefile to describe how to build your project).

Using make, you can ensure that the same steps are taken every time your software is built. Here's a makefile for a Kotlin project, that can build and execute, or run a simple test.

```
default:
    kotlinc Mean.kt -include-runtime -d out.jar

run:
    java -jar out.jar

test:
    java -jar out.jar 10 20

clean:
    rm out.jar
```

Limitations with Make

However, make may not be the best choice for large, complex projects.

1. **Build dependencies must be explicitly defined.**

- Libraries must be present on the build machine, manually maintained, and explicitly defined in your makefile.

2. **Make is tied to the underlying environment** of the build machine.

- It's difficult to completely isolate make's runtime behaviour from the underlying environment. e.g. \$LIB environment variable to track library location.

3. **Performance is poor.** Make doesn't scale well to large projects.

4. **The language itself isn't very expressive**, and has a number of inconsistencies.

5. **It's very difficult to fully automate** and integrate with other systems.

Gradle

There are a number of build systems on the market that attempt to address these problems. e.g. cmake, scons for C++, Ant or Maven for Java.

We're going to use Gradle with Kotlin.

- It's commonly used for large, complex Java and Kotlin projects.
- It handles all of our requirements, which is frankly, pretty impressive.
- It's the official build tool for Android builds, so you will need it for Android applications.
- It's declarative. You write Gradle build scripts in a DSL (Groovy or Kotlin), describing tasks to perform. Gradle figures out how to perform them.
- It handles multi-step builds and complex dependencies automatically! i.e. it tracks your libraries for you.

Gradle Commands

Gradle can be executed from the command-line. It supports a large range of commands.


\$ gradle help: shows available commands

\$ gradle init: create a new project and dir structure.

\$ gradle tasks: shows available tasks from build.gradle.

\$ gradle build: build project into build/

\$ gradle run: run from build/



As you've seen, the Gradle plugin for IntelliJ also makes these commands available in the IDE.

\$ gradle help

> Task :help

Welcome to Gradle 7.2.


To run a build, run gradle <task>

Gradle Projects

A Gradle project is simply a set of source files, resources and configuration files in a specific structure.

We can use Gradle to create a starting directory structure and build configuration files.

```
$ gradle init
Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2
...
```



The New Project wizard does the exact same thing.

Example: Hello Gradle

We can use Gradle to build and run a new project.

```
$ gradle build
BUILD SUCCESSFUL in 8s
8 actionable tasks: 8 executed
```

← Runs the “build” task

← Task output

```
$ gradle run
> Task :run
Hello world.
BUILD SUCCESSFUL in 1s
2 actionable tasks: 1 executed, 1 up-to-date
```

← Runs the “run” task

← Program output

Gradle is very “chatty”. This is actually very useful when debugging build issues.

You can use gradle tasks to see all supported actions. The available tasks will vary based on the type of project you create.

```
$ gradle tasks
```

```
> Task :tasks
```

```
-----  
Tasks runnable from root project  
-----
```

```
Application tasks  
-----
```

```
run - Runs this project as a JVM application
```

```
Build tasks  
-----
```

```
assemble - Assembles the outputs of this project.
```

```
build - Assembles and tests this project.
```

```
buildDependents - Assembles and tests this project and all projects that depend on it.
```

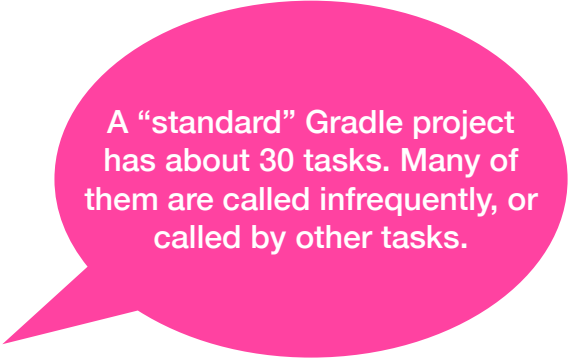
```
buildNeeded - Assembles and tests this project and all projects it depends on.
```

```
classes - Assembles main classes.
```

```
clean - Deletes the build directory.
```

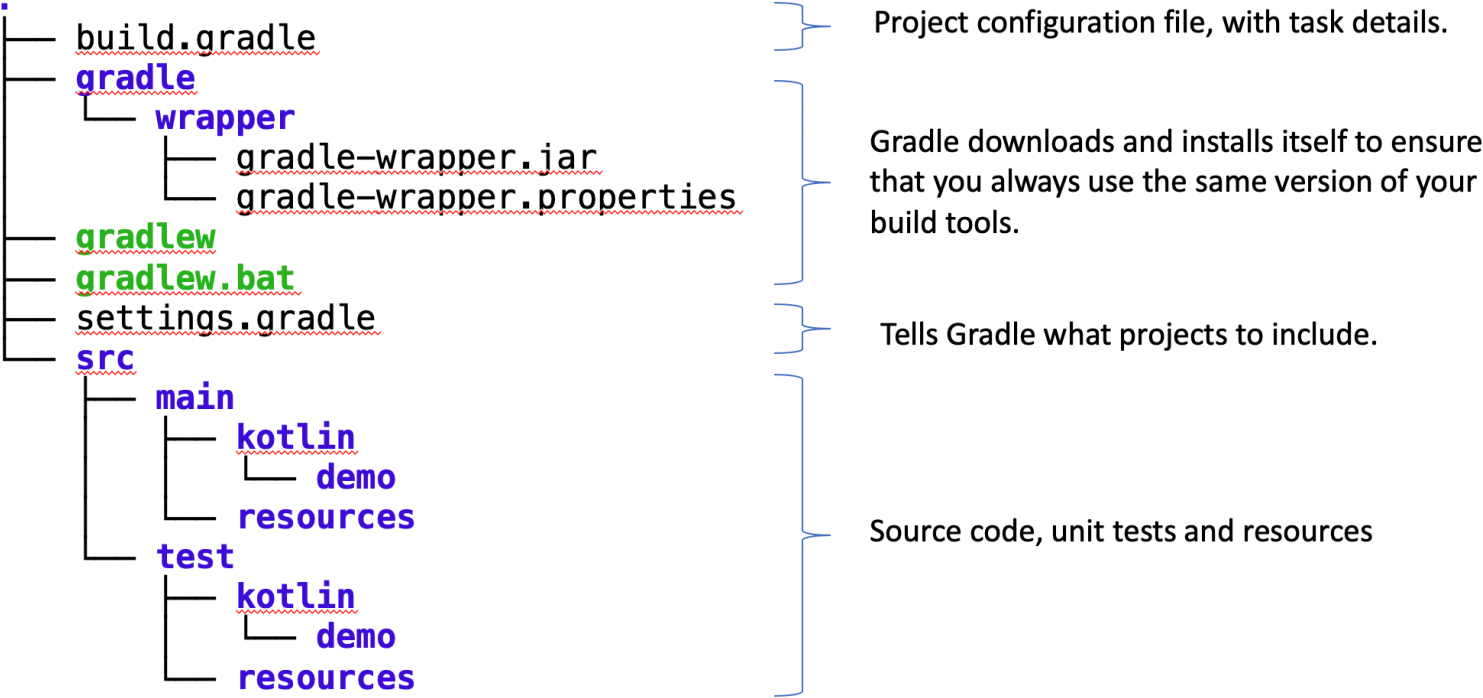
```
jar - Assembles a jar archive containing the main classes.
```

```
....
```



A “standard” Gradle project has about 30 tasks. Many of them are called infrequently, or called by other tasks.

A standard Gradle project directory is structured like this:



Gradle project structure

The **build.gradle** file contains our project configuration.

```
plugins {  
    // Kotlin JVM plugin to add support for Kotlin.  
    id 'org.jetbrains.kotlin.jvm' version '1.3.72'  
  
    // Application plugin for CLI applications.  
    id 'application'  
}
```

← Support for Kotlin language/builds

← Adds 'application' tasks e.g. run

```
repositories {  
    // Use jcenter for resolving dependencies.  
    // You can declare any Maven repository here.  
    jcenter()  
}
```

```
dependencies {  
    implementation platform('org.jetbrains.kotlin:kotlin-bom')  
    implementation 'org.jetbrains.kotlin:kotlin-stdlib-jdk8'  
    testImplementation 'org.jetbrains.kotlin:kotlin-test'  
    testImplementation 'org.jetbrains.kotlin:kotlin-test-junit'  
}  
application {  
    mainClassName = 'gradle.AppKt'  
}
```

← Libraries required for above

← Settings for 'application' plugin e.g. main class

build.gradle file

Benefits of Gradle

The build.gradle file contains information about your project, including the versions of all external libraries that you require. In this project file, you define how your project should be built:

- You define the versions of each tool that Gradle will use e.g. compiler version. This ensures that your toolchain is consistent.
- You define versions of each dependency e.g. library that your build requires. During the build, Gradle downloads and caches those libraries. This ensures that your dependencies remain consistent.
- Finally, Gradle has a wrapper around itself. You define the version of the build tools that you want to use, and when you run Gradle commands using the wrapper script, it will download and use the correct version of Gradle to actually create the builds. This ensures that your build tools are consistent.

Example: Calc.kt

```
package calc
```

```
fun main(args: Array<String>) {  
    try {  
        println(Calc().calculate(args))  
    } catch (e: Exception) {  
        print("Usage: number [+|-|*|/] number")  
    }  
}
```

 Main is not in a class

```
class Calc() {  
    fun calculate(args:Array<String>):Any {  
  
        if (args.size != 3) throw Exception("Invalid number of arguments")  
  
        val op1:String = args.get(0)  
        val operation:String = args.get(1)  
        val op2:String = args.get(2)  
  
        return(  
            when(operation) {  
                "+" -> op1.toInt() + op2.toInt()  
                "-" -> op1.toInt() - op2.toInt()  
                "*" -> op1.toInt() * op2.toInt()  
                "/" -> op1.toInt() / op2.toInt()  
                else -> "Unknown operator"  
            }  
        )  
    }  
}
```


Example: Calc.kt build

Let's migrate this code into a Gradle project.

1. Use Gradle to create the directory structure. Select "application" as the project type, and "Kotlin" as the language.

```
$ gradle init
```

```
Select type of project to generate:
```

```
1: basic
```

```
2: application
```

2. Copy the `calc.kt` file into `src/main`, and modify the `build.gradle` file to point to that source file.

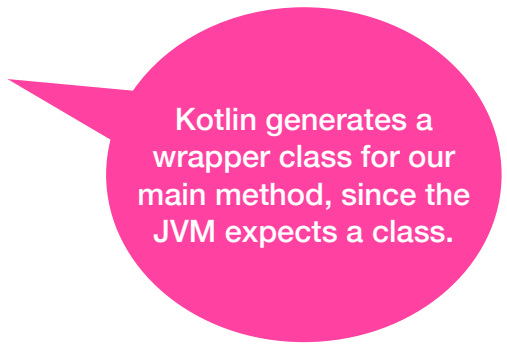
```
application {  
    // Main class for the application.  
    mainClassName = 'calc.CalcKt'  
}
```

3. Use gradle to make sure that it builds.

```
$ gradle build
```

```
...
```

```
BUILD SUCCESSFUL in 975ms
```



Kotlin generates a wrapper class for our main method, since the JVM expects a class.

4. If you use gradle run, you will see some unhelpful output:

```
$ gradle run
> Task :run
Usage: number [+|-|*|/] number
```

We need to pass arguments to the executable, which we can do with `--args`.

```
$ gradle run --args="2 + 3"
> Task :run
5
```

Organization

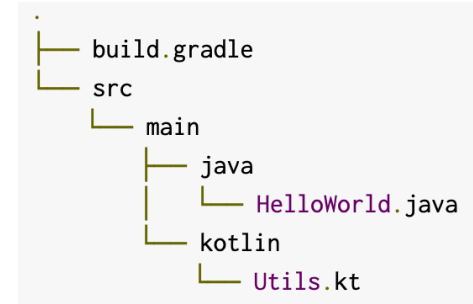
Structuring source code

Gradle provides some guidelines on how you should structure your source code. i.e. what sub-directories to create, how to organize your source files.

https://docs.gradle.org/current/userguide/organizing_gradle_projects.html

https://docs.gradle.org/current/userguide/directory_layout.html

1. **Separate source files by type.** The default Gradle structure splits up source files from configuration and resource files.
2. **Use standard conventions** as much as possible. Gradle and its plugins create default directories when you create a new project.



Gradle splits up source files by language.

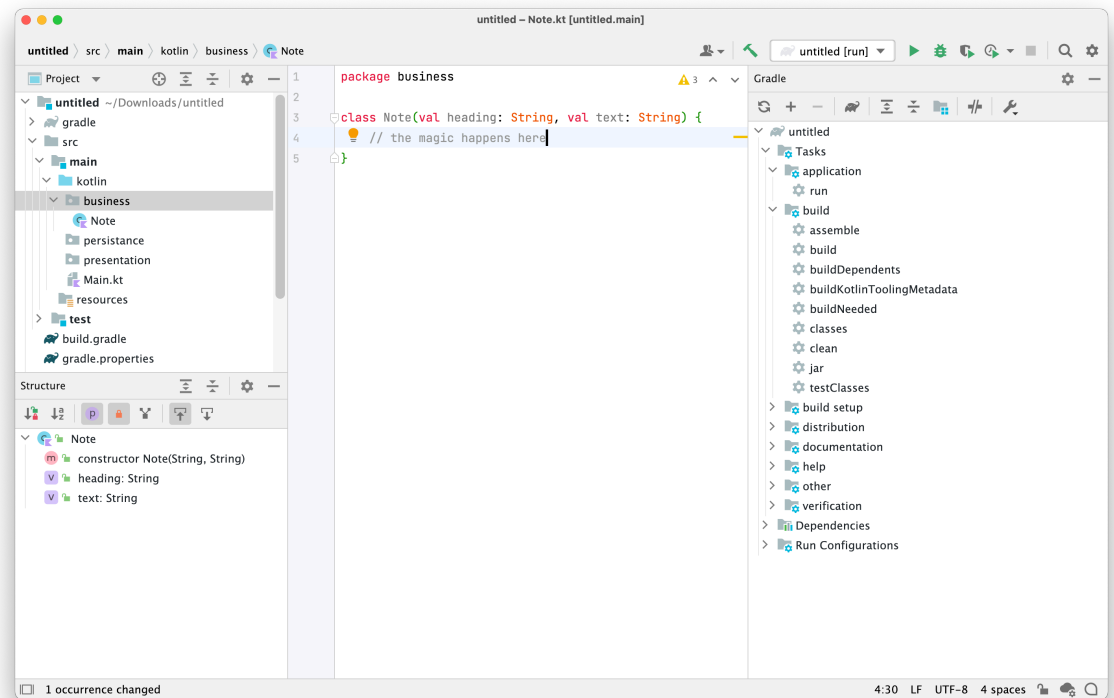
Structuring source code

You can further divide source code up by package, where a package represents a logical grouping of your files.

- /persistence (/models)
- /business (/controllers)
- /presentation (/views)

e.g. you are using a layered architecture, so you can further subdivide your source code by layer.

This also aids in testing, since you can write tests that target a specific layer (and focus on testing its interface).



Packages further group presentation, business and persistence layers.

Version Control

- Make sure to store your project in Git so that everyone can access it.
- For now, commit to the **Main** branch.
 - See the Git slides and videos for more information!

Tip:

Add a .gitignore to the top-level of your project, and in it, specify the build/ directory and any other temporary files or directories that you don't want saved.