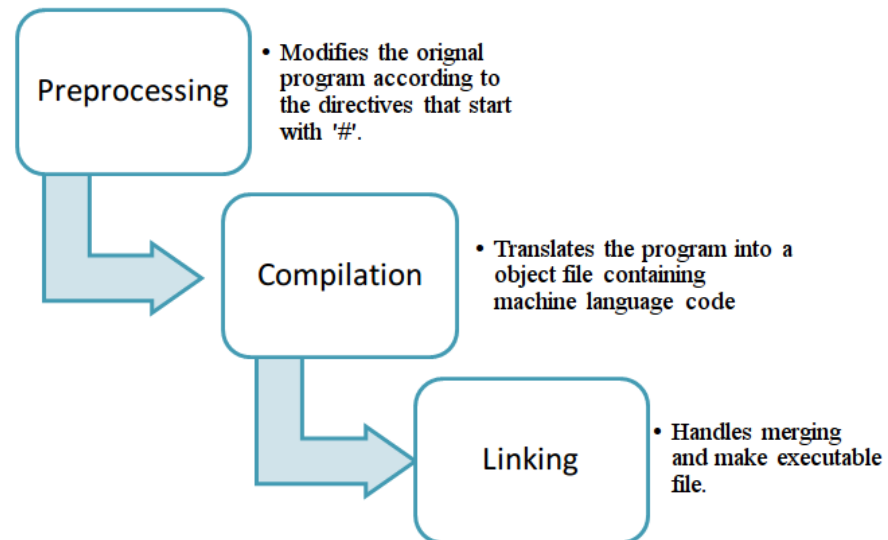


# Gradle

# 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, building software 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.

*For clarity: compiling refers to just compiling and linking. **Building** refers to the complete set of steps.*

Performing these steps manually is error prone, and very time-consuming. The ideal system would be automated and have the following characteristics.

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**.
  - Import the newest version of a library when it's published.
  - Rebuild and test when someone commits to the Test branch.
5. Tasks could be **scheduled**.
  - 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 one of the most widely used build systems, which allows you to script your 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.

For small or relatively simple projects, make is a perfectly reasonable choice. It's easy to setup, and is pre-installed on many systems.

**Makefiles** are language-independent, and we can use them to ensure consistency in how builds are performed.

We aim for a system that will provide the same results every time we build our software.

default:

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

run:

```
java -jar out.jar
```

test:

```
java -jar out.jar 10 20
```

```
java -jar out.jar 1 2 3 4
```

clean:

```
rm out.jar
```

# Limitations with Make

However, make has limitations and may not be the best choice for large or more 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 fragile and tied to the underlying environment of the build machine.
  - e.g. \$LIB environment variables to track libraries on the filesystem
  - It's difficult to completely isolate make's runtime behaviour from the underlying environment.
3. Performance is poor. Make doesn't scale well to large projects.
4. Its language isn't very expressive, and has a number of inconsistencies.
5. It's very difficult to fully automate and integrate with other systems.
  - What if we want to update a live dashboard with build results? Email people when the build fails?
  - Make is fine for compiling, but isn't designed to handle any other steps.



# Gradle

There are a number of build systems on the market that attempt to address these problems. They are often programming-language or toolchain dependent.

- C++: CMake, Scons, Premake
- Java: Ant, Maven, Gradle

We're going to use Gradle in this course. Why?

- It's commonly used for large, complex 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.

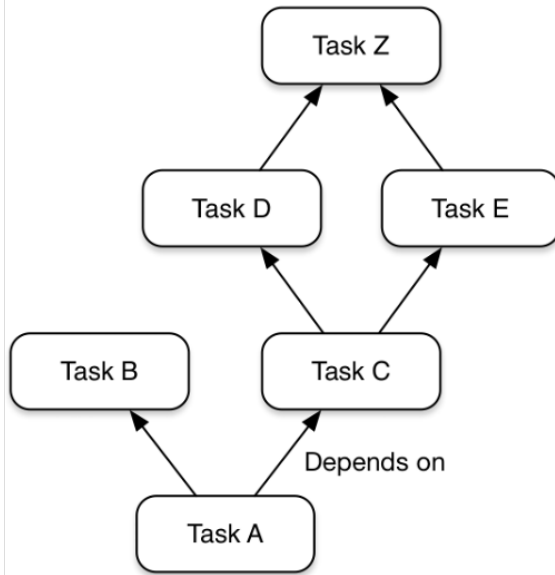
**Gradle** is a build system designed for large-scale projects. It's cross-platform and language agnostic.

- Gradle is the engine that drives the build.
- Groovy is a DSL scripting language that Gradle has adopted.

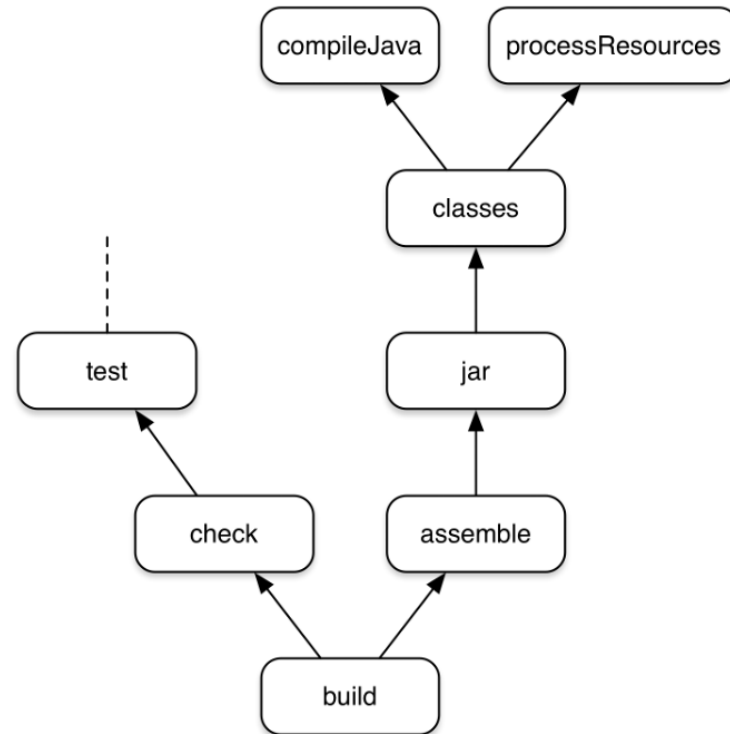
You write Gradle build scripts in a DSL (Groovy), describing tasks to perform. Gradle figures out how to perform them.

Gradle handles dependency management, multi-step build tasks and manages complex dependencies automatically!

Generic task graph



Partial task graph for a standard Java build



Task graph for a typical Java build

# Gradle Commands

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

- **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/

```
$ gradle help
> Task :help
Welcome to Gradle 6.4.1.
To run a build, run gradle <task>
```

# Gradle Projects

A Gradle project is simply a set of source files, resources and configuration files structured so that Gradle can build it.

Gradle projects require a very specific directory structure. We could create this by hand, but for now let's use Gradle to create a starting directory structure and build configuration file that we can modify.

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

# Example: Hello Gradle

The starter project that we just created includes some basic code. Let's use Gradle to build and run it.

```
$ 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.
```

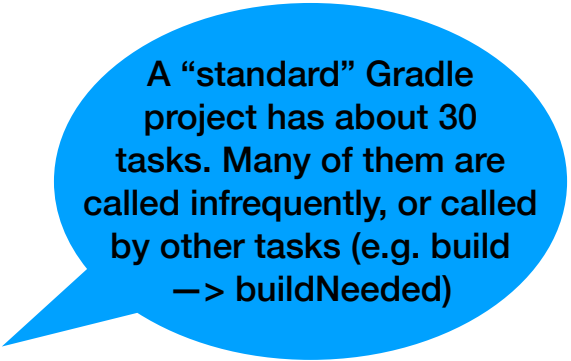
```
buildNeeded - Assembles and tests this project and all projects it depends on.
```

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

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

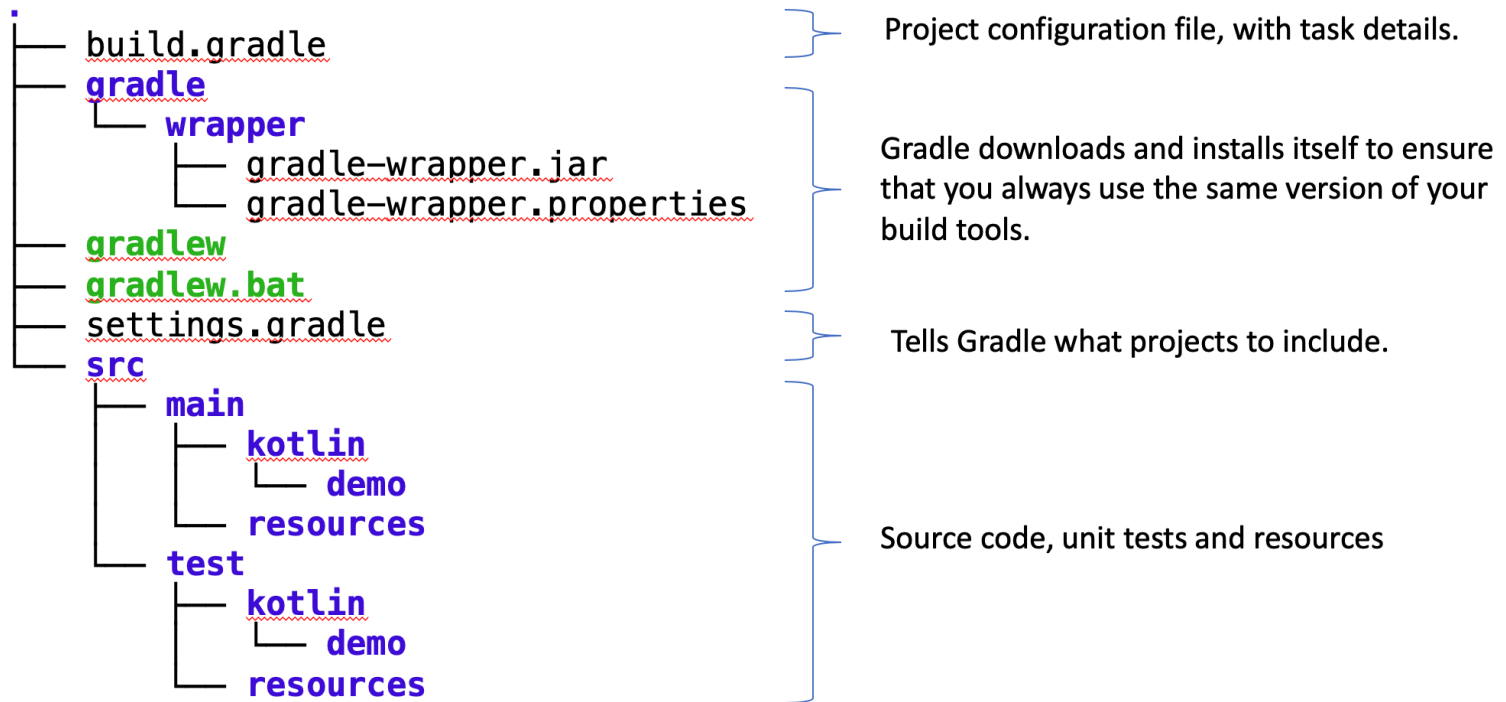
```
jar - Assembles a jar archive containing the main classes.
```

```
....
```



A “standard” Gradle project has about 30 tasks. Many of them are called infrequently, or called by other tasks (e.g. build → buildNeeded)

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")  
    }  
}  
  
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.

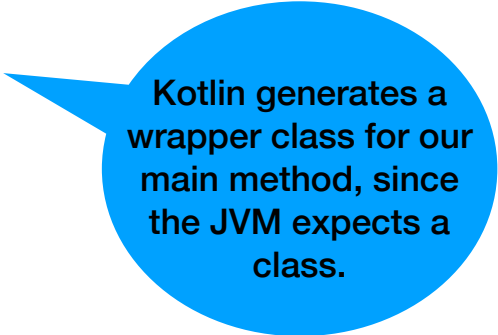
```
application {  
    // Main class for the application.  
    mainClassName = 'calc.CalcKt'  
}
```

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
```