# Contributing

This is for code contributions. For translations, see [TRANSLATING](TRANSLATING.md).

## Basic Guidelines

#### Use an IDE.

Specifically, IntelliJ IDEA. Download the (free) Community Edition of it [here](https://www.jetbrains.com/idea/download/). Some people use other tools, like VS Code, but I would personally not recommend them for Java development.

#### Always test your changes.

Do not submit something without at least running the game to see if it compiles.

If you are submitting a new block, make sure it has a name and description, and that it works correctly in-game. If you are changing existing block mechanics, test them out first.

#### Do not make large changes before discussing them first.

If you are interested in adding a large mechanic/feature or changing large amounts of code, first contact me (Anuken) via [Discord](https://discord.gg/mindustry) (preferred method) or via e-mail (\*anukendev@gmail.com\*).

For most changes, this should not be necessary. I just want to know if you're doing something big so I can offer advice and/or make sure you're not wasting your time on it.

## Style Guidelines

#### Follow the formatting guidelines.

This means:

- No spaces around parentheses: `if(condition){`, `SomeType s = (SomeType)object`

- Same-line braces.

- 4 spaces indentation

- `camelCase`, \*\*even for constants or enums\*\*. Why? Because `SCREAMING\_CASE` is ugly, annoying to type and does not achieve anything useful. Constants are \*less\* dangerous than variables, not more.

- No underscores for anything. (Yes, I know `Bindings` violates this principle, but that's for legacy reasons and really should be cleaned up some day)

- Do not use braceless `if/else` statements. `if(x) statement else statement2` should \*\*never\*\* be done. In very specific situations, having braceless if-statements on one line is allowed: `if(cond) return;` would be valid.

Import [this style file](.github/Mindustry-CodeStyle-IJ.xml) into IntelliJ to get correct formatting when developing Mindustry.

#### Do not use incompatible Java features (java.util.function, java.awt).

Android and RoboVM (iOS) do not support many of Java 8's features, such as the packages `java.util.function`, `java.util.stream` or `forEach` in collections. Do not use these in your code.

If you need to use functional interfaces, use the ones in `arc.func`, which are more or less the same with different naming schemes.

The same applies to any class \*outside\* of the standard `java.[n]io` / `java.net` / `java.util` packages: Most of them are not supported.

`java.awt` is one of these packages: do not use it, ever. It is not supported on any platform, even desktop - the entire package is removed during JRE minimization.

In general, if you are using IntelliJ, you should be warned about platform incompatiblities.

#### Use `arc` collections and classes when possible.

Instead of using `java.util.List`, `java.util.HashMap`, and other standard Java collections, use `Array`, `ObjectMap` and other equivalents from `arc.struct`.

Why? Because that's what the rest of the codebase uses, and the standard collections have a lot of cruft and usability issues associated with them.

In the rare case that concurrency is required, you may use the standard Java classes for that purpose (e.g. `CopyOnWriteArrayList`).

What you'll usually need to change:

- `HashSet` -> `ObjectSet`

- `HashMap` -> `ObjectMap`

- `List` / `ArrayList` / `Stack` -> `Array`

- `java.util.Queue` -> `arc.struct.Queue`

- \*Many others\*

#### Avoid boxed types (Integer, Boolean)

Never create variables or collections with boxed types `Array<Integer>` or `ObjectMap<Integer, ...>`. Use the collections specialized for this task, e.g. `IntArray` and `IntMap`.

#### Do not allocate anything if possible.

Never allocate `new` objects in the main loop. If you absolutely require new objects, use `Pools` to obtain and free object instances.

Otherwise, use the `Tmp` variables for things like vector/shape operations, or create `static` variables for re-use.

If using a list, make it a static variable and clear it every time it is used. Re-use as much as possible.

#### Avoid bloated code and unnecessary getters/setters.

This is situational, but in essence what it means is to avoid using any sort of getters and setters unless absolutely necessary. Public or protected fields should suffice for most things.

If something needs to be encapsulated in the future, IntelliJ can handle it with a few clicks.

#### Do not create methods unless necessary.

Unless a block of code is very large or used in more than 1-2 places, don't split it up into a separate method. Making unnecessary methods only creates confusion, and may slightly decrease performance.