The Design Recipe

On this page

The Design Recipe

This page will contain information about the design recipe, and will evolve as the course progresses.

Multiple Audiences

Always Remember: Code has multiple audiences. The ones that we care about in particular are computers and programmers.

Computers

Computers need to be able to run your code. They have a very limited understanding of the world, and will mostly do whatever you tell them, whether it makes sense or not. We tell them what to do in a language that is made for them, and so it may not always be obvious for humans what a particular piece of code means in terms that we have any intuition for.

The good thing is that these kinds of languages are very precise, and that in turn enables us to write tooling around them. One such tool is called a *type checker*. Java uses such a type checker to judge whether a program you wrote makes sense or not with respect to a very crude, but largely effective understanding of the world. In essence, you need to label every piece of data that you handle in code with a *type*, which is a descriptor for all possib values that that piece of data could have. This is often exceedingly coarse, for example because Java only considers one general type of text data, `String`, which can be any arbitrarily long combination of symbols, including the empty `String`. Java does not provide a way to label a piece of data as being only one of a smaller set of possible strings, such as `{ "a", "b", "hello!" }`, even if you as the programmer know that these are the only possible values a particular variable in your program could have at a given point. In order for your program to pass the *type checker*, Java therefore requires your code to be able **all** possible Strings, not just the ones you know could actually show up.

In exchange, Java guarantees to you that certain errors are not going to happen in your program, because it has thoroughly checked everything and conservatively excluded a number of potential errors. That does not mean that your program is error-free. There are lots of potential errors that Java's type checker cannot catch. One example of this are divisions by zero, since during type-checking, Java only knows that a number is an `int`, not what its possible values might be.

All of that is to say that Java may be too strict in some situations and too lenient in the other, and overall, language designers aim for a reasonable middle ground. But just because we are a bit limited in what how we can express ourselves to a computer, does not

mean that we cannot give more information to about our code to other people who might be looking at it.

Programmers

Programmers - and that includes you, in particular you a few days from whenever you wrote a particular piece of code - can in principle mostly read code and see what it does on a very local level. But programs can get quite big, and jumping around between definitions, all the while thinking of all potential edge cases, is arduous and error-prone.

Additionally, if code is all there is, then we are reading information on the same coarse level as our programming language's tooling is. But we already established that that level is often not precise enough to express what we really intend to say. Therefore, we need some other way to add additional information that we cannot express in Java code directly.

To do that, we use *code comments*, which are pieces of text in code that is only there for other humans, and completely ignored by the computer. In Java, there are several ways to write comments:

```
// two slashes make any text on the rest of the line a comment.
println("Hello World"); // this means they can follow some code

/* A slash / followed by a star * starts a comment block.
This block continues until a * is immediately followed by a slash,
across lines */
println("Hello World!" /* this means that such a comment
can also appear in the middle of some code */);
println("/* however, no comments can appear within strings*/");

/**
 * Finally, there is a special kind of block comment meant
 * to document definitions. Editors often insert the leading
 * stars * for you when you switch to a new line.
 * This block comment simply starts with a slash / and two
 * starts * instead of just one.
 * We use this kind of comment *a lot*!
 */
```

The Design Recipe we use in this course requires you to mostly write the last kind of block comment in many places, and follow a particular style that includes particular pieces of information. Oftentimes, that requires you to write down things that you think are obvious, and we will generally err on the side of writing too many comments than too few. In practice, programmers often write fewer comments, though big companies like Google have their own rules about them and do require a substantial amount of them - because

communicating well with other programmers in the same company is vitally important for software quality.

The commenting style here is meant to prompt you think about the right questions while you are writing code. As you do this more and more, you will learn to automatically think about these questions, even once you don't have to strictly follow the Design Recipe anymore. For that reason, it is best if you actually follow the steps as outlined here, and not add the necessary comments as an afterthought.

The Wishlist

The wish list is a list of functions that you still need to design. Each such function has a name, a purpose statement, and at least an idea of a signature. Depending on your problem, you will start out with at least one, but possibly several functions. For an interactive program, that may be the `main` function, for a world program, that may additionally be the functions to draw, and, potentially, step, and react to keyboard and mouse events. For some assignments that require certain functions, it will be those functions.

As you design those initial functions, you will often encounter the need for one or more helper functions, which at first you just add to the wishlist. Once you are done with designing a particular function, you continue with the next function from the wishlist, until the wishlist is eventually empty.

Step 1: Problem Analysis and Data Design

In this first step, you mainly consider the next item on your wishlist, but can also consider all the items of the wishlist as whole. Key questions you should ask yourself are:

- What kind of information will you need in order to be able to implement these items?
- What information will you be given, what information do you need to assemble?
- How should this all be represented as data?
- What are the kinds of constraints on this data that your language (in this case Java) enforces, and what are the constraints that matter to programmers and users?

Based on the answers to these question, you may need to come up with one or more data definitions, as described below.

Templates

All data definitions except for basic data come with (code) templates. These templates express a key slogan that we follow in this class:

> **The shape of the data determines the shape of the program.**

As such, for the non-basic data definitions, there is a particular way you should write code whenever you are analyzing a piece of such data. The templates guide you in this effort. Each template is easily derived from its corresponding data definition, and you should add it as a commented-out code block right below that data definition. You will see some examples below.

Basic Data

Basic data are values of the basic Java data types:

```
boolean
int
long
float
double
char
String
```

Additionally, the standard library defines the following opaque types, which also count as basic data:

```
Date
DateTime
Colour
Image
```

In many cases, you do not have to do anything special for basic data. The Java types already exist, and basic corresponding human interpretations do, too.

However, in some cases, you may use a basic type in a restricted way, and/or with a certain interpretation. You can mention this restriction every time you use the type, or you can create a basic type definition that you can refer back to. When you make a basic type definition, you need to include at least some examples.

Interpretations

If you define a basic data type on your own, you always have to give it an interpretation. An interpretation is a comment text that says a little about what values of this type are supposed to represent. For example:

```
/**
 * A Temperature is a double that measures some temperature
 * in degrees Celsius (>= -273)
 * Examples:
```

```
 * -20C, 0C, 24C, 100C
 */
```

In particular for types that represent measures of some thing, it is important to clearly state the unit. It makes a huge difference if the duration of something is measured in seconds, minutes, hours, days, or years! Similarly, it makes a huge difference whether you measure a space-probe's impulse in pound-force seconds or newton-seconds!

Restrictions

Most basic types allow for huge numbers of potential values (with the exception of `boolean`, which has just two). Some times, a function is not well-defined on all of them. An standard case is integer division, which is not defined for the case where 0 is the divisor (trying to do such a division will crash your program). Java's types are not precise enough to capture this, but you can document your intentions that a number not be zero by writing this in your relevant comments. Basic data type definitions give you a shorthand for such comments.

```
/**
 * A NonZeroInt is an integer that is no 0.
 * Examples:
 * -25235, -15, 1, 1336
 */
```

Another common restriction is that number types only represent nonnegative numbers, again particularly with integers:

```
/**
 * An ItemCount is an int that represents a count
 * of items, which is nonnegative.
 * Examples:
 * 0, 1, 2, 15, 19, 1525
 */
```

A similar restriction might appear with Strings:

```
/**
 * A Name is a String that consists of printable characters
 * and whose length (without whitespace) is greater than 0.
 * Examples:
 * "Hugo", "Lisa", "M", "Grepnor, Dark Lord Of The Universe"
 */

/**
```

```
 * A UID is a String of exactly 8 characters that starts with
 * a lowercase "u" and continues with 7 decimal digits.
 * Examples"
 * "u1234567", "u7654321", "u1122334"
 */
```

Enumerations

Sometimes, a relevant piece of information is which of a specific number of specific cases applies. The preferred way to define this kind of data is using Java's `enum`s, but in some cases, it may also make sense to use values from a single basic data type. Because enumerations specifically list all possible cases, it is not necessary to list examples.

Java Enumerations

A Java enumeration has the following form:

```
enum [Name] { [CASE1], [CASE2], ... }
```

> In *Functional Java*, only the list of case names is allowed between the curly braces `{}` (this is mostly for people who already know Java from elsewhere).

You need to document the enumeration with an interpretation for the enumeration as a whole, and an interpretation for each case. For example, to define the potential states of a traffic light, we'd write:

```
/**
 * A TrafficLightState represents the state of a traffic light
 * and how it instructs traffic to flow.
 */
enum TrafficLightState {
    /** Traffic must stop */
    RED,
    /** Traffic must stop if able */
    YELLOW,
    /** Traffic can go ahead */
    GREEN
}
```

At this point, you can write `TrafficLightState.RED`, `TrafficLightState.YELLOW`, and `TrafficLightState.GREEN` elsewhere in your Java code to produce the

corresponding value of type `TrafficLightState`.

Templates

If you are given an enumeration value, the main question is which case of the enumeration currently applies. Therefore, a template for enumeration data is always about distinguishing between the cases of the enumeration. If you have an enumeration `MyEnum` with cases `CASE1`, `CASE2`, ... `CASEn`, then the template looks as follows:

```
// {
//     ...
//     return ... switch(myEnum) {
//         case CASE1 -> ...;
//         case CASE2 -> ...;
//         [...]
//         case CASEn -> ...;
//     } ...;
// }
```

For example, if you are given a value of type `TrafficLightState`, the main question is usually which concrete state it actually is. Accordingly, the template for `TrafficLightState` looks as follows:

```
// {
//     ...
//     return ... switch(trafficLightState) {
//         case RED -> ...;
//         case YELLOW -> ...;
//         case GREEN -> ...;
//     } ...;
// }
```

The `...`s mark places in the template where you might fill in other code. In most cases, you will use the `...`s after the arrows `->`.

Put the template below your data definition. For `TrafficLightState`, the full definition will therefore look as follows:

```
/**
 * A TrafficLightState represents the state of a traffic light
 * and how it instructs traffic to flow.
 */
enum TrafficLightState {
    /** Traffic must stop */
```

```
    RED,
    /** Traffic must stop if able */
    YELLOW,
    /** Traffic can go ahead */
    GREEN
}

// Template:
// {
//     ...
//     return ... switch(trafficLightState) {
//         case RED -> ...;
//         case YELLOW -> ...;
//         case GREEN -> ...;
//     } ...;
// }
```

Basic Data Enumerations

In some cases, it makes sense to use values of a basic data type instead of Java enumerations to define an enumeration. All values need to be of the same basic data type. The overall principles are the same as for the Java-enumeration-based version, but there is no actual Java code to write for the definition:

```
/**
 * A TrafficLightState is a String that represents the state
 * of a traffic light and how it instructs traffic to flow.
 * It is one of:
 * - "red" - Traffic must stop
 * - "yellow" - Traffic must stop if able
 * - "green" - Traffic can go ahead
 */
```

Templates

The corresponding template needs to deal with the fact that Java only knows that `TrafficLightState` s are `String` s, which from Java's perspective could be arbitrary Strings. Therefore, the `switch` -expression in the template needs to include a `default` case. If this `default` case ever applies, that indicates an error in the program. The standard library has a special function, called `DefaultCaseError` , which you can use here to make sure the program aborts and signals an error in this case.

```
// Template:
// {
```

```
//     ...
//     return ... switch(trafficLightState) {
//         case "red" -> ...;
//         case "yellow" -> ...;
//         case "green" -> ...;
//         default -> DefaultCaseError("Bad traffic light state!");
//     } ...;
// }
```

Records

Records are for representing cases where information consists of a group of smaller pieces of information, and it is always the same smaller pieces. To define a record in Java, you write:

```
record [Name]([Type1] [fieldName1], [Type2] [fieldName2], ...) {}
```

That is a record has a type name `[Name]` and some number of fields of names `[fieldName1]`, `[fieldName2]`, `...` with corresponding types `[Type1]`, `[Type2]`, `....`

> The curly braces `{}` are there because Java requires them, but in *Functional Java* they should never have any content.

To create a new value of that type, you write:

```
new [Name]([value-for-fieldName1], [value-for-fieldName2], ...)
```

If you have a value of that type, you can access its fields as follows:

```
[value-of-type-Name].[fieldName{1/2...}]()
```

As before, you need to provide interpretations of both the type as a whole and its parts, as well as examples. The interpretations of the fields of the record use *javadoc* syntax to refer to the fields, that is, the relevant lines start with `@param [fieldName]`. For example, you might consider the following representation of students:

```
/**
 * A Student is a record that contains key information about an
 * ANU student.
 * Examples:
```

```
 * - Lisa Studywoman, u1234567, BAC
 * - Paul Masterson, u7654321, MCOMP
 * @param name - the name of the student, a non-empty String
 * @param uid - a UID, indicating the student's university ID
 * @param program - a String describing the student's degree program
 */
record Student(String name, String uid, String program) {}
```

This example also illustrates the use of basic data definitions. We use the definition of `UID` from the examples up in the section on basic data definitions to refer to the global definition that UIDs are 8-character Strings starting with a lowercase “u” and otherwise only containing digits. We could have also referred to the definition of `Name` above, but instead the example illustrates that you can also add restrictions to the description of a field directly, in this case that the `name` field must contain a non-empty String.

With this definition, you can create a new `Student` value with:

```
new Student("Lisa Studywoman", "u1234567", "BAC")
```

And given a value of type `Student` (in this example named `student`), you can access it's field with the following three accessors:

```
student.name()
student.uid()
student.program()
```

More Complicated Constraints

Sometimes, restrictions on the values of fields are not limited to an individual field on its own. In this case, you need to specifiy a more global *invariant* on the data type. For this, we use the *javadoc* field `@implSpec` after the `@param` lines.

For example, some people only have a single name that does not neatly fit into the concept of first and last names. But a name overall should still be non-empty. So if you keep track of first and last names separately, but require the overall non-emptiness of the combined name, you can specify that as follows:

```
/**
 * A Person is represented by a record that contains basic
 * personal attributes.
 * Examples:
 * - Jane Smith, age 21, 60'' tall, weighs 60kg
 * - Winston Crikey, age 78, 72'' tall, weighs 80kg
 * @param firstName - the first/given name of a person, or an
```

```
 *                   empty String if no such name applies
 * @param lastName - the last/family name of a person, or an
 *                   empty String if no such name applies
 * @param age - the age of a person in years (non-negative)
 * @param height - the height of a person in inches (non-negative)
 * @param weight - the weight of a person in kg (non-negative)
 * @implSpec Invariant: at least one of firstName or lastName is
 * not an empty String and contains some non-whitespace characters
 */
record Person(String firstName,
              String lastName,
              int age,
              int height,
              int weight) {}
```

Invariant is the term for a condition that always needs to be satisfied. Anyone who creates a value of the record must do so in a manner that makes the invariant true, and anyone receiving a value of a record can rely on the invariant being true.

Templates

The template for records is admittedly a bit boring. It simply refers to the fact that you can access all the fields of a record:

```
// { ... myRecord.[fieldName1]() ... myRecord.[fieldName2]() ... [..
```

For our `Student` example, the template would look as follows:

```
// { ... student.name() ... student.uid() ... student.program() ... }
```

Just because the fields appear in the template does not mean you have to use them, or use them just once, or in the same order. You can copy, reorder, and delete any field accesses in this template. This template is an exception to the Apply Template design strategy as explained below: you can treat it as part of a Combining Functions design strategy instead of a template application.

General Itemizations

Enumerations and records can be combined in a very useful way: you may both need to be able to distinguish different cases, and depending on which case it is, there might be different information associated with it. For example, we might want to talk about geometric shapes, where circles are defined by their radius, rectangles by their width and height, and orthogonal triangles by the lengths of their legs. Each of them can on their own

be expressed as a `record`, but that would just yield three different types that Java keeps distinct, so you would never have to distinguish between circles, rectangles, and triangles, because you could not mix them in the first place.

Luckily, Java provides us with a mechanism to express that we want to mix a certain set of records in a way that lets us later distinguish them again, called *sealed interfaces*. These are a special case of a general and much more important Java feature - *interfaces* - which will come in when we switch to regular Java later. For now, only *sealed interfaces* are allowed.

A *General Itemization* is a data definition that consists of several distinct cases, each of which associated with their own data. They are defined in two steps:

- first, a sealed interface lists exactly all the cases we consider, and provides us with a Java type name that lets us treat them all the same.
- second, each case has its own record definition, similar to the definitions of records from before

The Sealed Interface

A sealed interface declaration has the following form:

```
sealed interface [Name] permits [Case1], [Case2], ... {}
```

where `[Name]` is the name of the general itemization data type, and `[Case1], [Case2], ...` are the names of the individual cases.

> The curly braces `{}` are there because Java requires them, but in *Functional Java* they should never have any content.

A sealed interface needs to be documented with the overall interpretation of the data type, as well as a quick summary of the potential cases. We'll leave the examples and field descriptions to the records themselves.

For example, if we want to model shapes as above, we might write:

```
/**
 * A Shape represents a geometric shape and is one of:
 * - A Circle, representing circular shapes
 * - A Rectangle, representing rectangular shapes
 * - An OrthogonalTriangle, representing shapes that are
 *                          orthogonal triangles
 */
```

```
sealed interface Shape
  permits Circle, Rectangle, OrthogonalTriangle {}
```

The case documentation here is a bit verbose, and in obvious cases like this one, you can omit the parts that start with “representing”. In general, it needs to be clear what each potential case of an itemization is supposed to represent.

The Records

Defining the cases works almost exactly the same as defining the records on their own. The main exceptions are as follows:

- you do not need to provide a template for the records if you never use them on your own (which is the normal case)
- you need to link the records back to the sealed interface, as shown below

Java insists that a `record` mentioned as a case of a `sealed interface` explicitly mention this in their own definition, too. Thus, the general scheme for the code defining records is as follows:

```
record [Name]([Type1] [fieldName1], [Type2] [fieldName2], ...)
  implements [Name-of-sealed-interface] {}
```

This is just like for normal records, except that we add `implements [Name-of-sealed-interface]` between the field list and the empty curly braces `{}`.

Therefore, in our example, the definitions of the three records mentioned above would looks as follows:

```
/**
 * A Circle is a Shape characterized by its radius.
 * Examples:
 * - A circle of radius 1.0
 * - A circle of radius 0.5
 * @param radius – the radius of the circle, in whatever units
 * the program uses for Shapes, non-negative
 */
record Circle(double radius) implements Shape {}

/**
 * A Rectangle is a Shape characterized by its width and height.
 * Examples:
 * - A rectangle of width 1.0 and height 0.5
 * - A rectangle of width 2.0 and height 2.0
```

```
 * @param width – the width of the rectangle, in whatever units
 * the program uses for Shapes, non-negative
 * @param height – the height of the rectangle, in whatever units
 * the program uses for Shapes, non-negative
 */
record Rectangle(double width, double height) implements Shape {}

/**
 * An Orthogonal Triangle is a Shape characterized by the lengths
 * of its two legs.
 * Examples:
 * - An orthogonal triangle whose left leg length is 3.0 and
 *   whose right leg length is 4.0
 * - An orthogonal triangle whose legs both have length 1.0
 * @param leftLeg – the length of the left leg, in whatever units
 * the program uses for Shapes
 * @param rightLeg – the length of the right leg, in whatever
 * units the program uses for Shapes
 */
record OrthogonalTriangle(double leftLeg, double rightLeg)
    implements Shape {}
```

Templates

Being a collection of multiple different cases, the template for general itemizations contains a `switch`-expression, similar to the one for enumerations. However, as the case are more complex than for enumerations, so is the template. It's general form is:

```
// {
//     ...
//     return ... switch(myItemization) {
//         case [Case1](var [cse1fld1], var [cse1fld2], ...) -> ...;
//         case [Case2](var [cse2fld1], var [cse2fld2], ...) -> ...;
//         [...]
//     } ...;
// }
```

That is, we use the names of the records as `[Case1]`, `[Case2]`, `...`, and provide variable names that will contain their fields for each field in the respective case. These variables are accessible on the right-hand side of the arrow `->` for the respective case, and contain the values of the corresponding fields in the given record.

For our `Shape` example, the template looks as follows:

```
// {
//     ...
//     return ... switch(shape) {
//         case Circle(var radius) -> ...;
//         case Rectangle(var width, var height) -> ...;
//         case OrthogonalTriangle(var leftLeg, var rightLeg) -> ...;
//     } ...;
// }
```

Altogether, the data definition for shapes is as follows:

```
/**
 * A Shape represents a geometric shape and is one of:
 * - A Circle, representing circular shapes
 * - A Rectangle, representing rectangular shapes
 * - An OrthogonalTriangle, representing shapes that are
 *                          orthogonal triangles
 */
sealed interface Shape
  permits Circle, Rectangle, OrthogonalTriangle {}


/**
 * A Circle is a Shape characterized by its radius.
 * Examples:
 * - A circle of radius 1.0
 * - A circle of radius 0.5
 * @param radius – the radius of the circle, in whatever units
 * the program uses for Shapes, non-negative
 */
record Circle(double radius) implements Shape {}

/**
 * A Rectangle is a Shape characterized by its width and height.
 * Examples:
 * - A rectangle of width 1.0 and height 0.5
 * - A rectangle of width 2.0 and height 2.0
 * @param width – the width of the rectangle, in whatever units
 * the program uses for Shapes, non-negative
 * @param height – the height of the rectangle, in whatever units
 * the program uses for Shapes, non-negative
 */
```

```
record Rectangle(double width, double height) implements Shape {}

/**
 * An Orthogonal Triangle is a Shape characterized by the lengths
 * of its two legs.
 * Examples:
 * - An orthogonal triangle whose left leg length is 3.0 and
 *   whose right leg length is 4.0
 * - An orthogonal triangle whose legs both have length 1.0
 * @param leftLeg – the length of the left leg, in whatever units
 * the program uses for Shapes
 * @param rightLeg – the length of the right leg, in whatever
 * units the program uses for Shapes
 */
record OrthogonalTriangle(double leftLeg, double rightLeg)
    implements Shape {}

// Template:
// {
//     ...
//     return ... switch(shape) {
//         case Circle(var radius) -> ...;
//         case Rectangle(var width, var height) -> ...;
//         case OrthogonalTriangle(var leftLeg, var rightLeg) -> ...
//     } ...;
// }
```

Functions

To use lambda abstractions, you need to be able to type them. For this purpose (Functional) Java provides standard Function types:

- `Supplier<R>` -for functions that take no arguments an return a value of type `R` as output
- `Function<A, R>` -for functions that take a single value of type `A` as input and return a value of type `R` as output
- `BiFunction<A1, A2, R>` -for functions that take two arguments of of types `A1` and `A2`, respectively, as input and return a value of type `R` as output
- `Predicate<A>` -for functions that take a single value of type `A` as input and return a `boolean`
- `BiPredicate<A1, A2>` -for functions that take two arguments of of types `A1` and `A2`, respectively, as input and return a `boolean` Additionally, there is the type

`Runnable` for functions that take no arguments and have return type `void`. For our purposes, this is only useful for testing.

For more information on this, see the section on lambdas in the Function Java Documentation.

Given that the Java types already exist, function data definitions work similarly to basic data definitions, in that they only consist of comments. The main difference is that they essentially need the first three steps of the design recipe, adjusted for the generality of the data definition: depending on how big the class of functions that you want to accept is, you may need to specify more or fewer restrictions, and also depending on the generality, you may or may not be able to give concrete examples. As a syntactic difference, for the human signature part of the design recipe for this data definition, we omit the `@` symbol in the `javadoc` markers. For example:

```
/**
 * An IntComparator is a BiFunction<Integer, Integer, Integer>,
 * which returns the result of comparing two numbers.
 * param left - an integer to compare
 * param right - an integer to compare
 * returns 0 if left equals right, a negative integer if left
 *   should count as "less" than right, and a positive integer
 *   if left should count as "greater" than right
 */
```

A possible instance of such a function type is `(left, right) -> -Compare(left, right)`, which inverts the result of the standar comparison function. An example use of something like this is to invert the sorting order of a list. Because of the generality of the function type, it is fine to leave out examples.

Another example can use some concrete examples:

```
/**
 * An IncreasingSorter is a
 * Function<ConsList<Integer>, ConstList<Integer>>,
 * which returns a version of the given list that is sorted
 * in ascending order.
 * Examples:
 * 1, 2, 3, 7, 4 -> 1, 2, 3, 4, 7
 * 5, 4, 3, 2, 1 -> 1, 2, 3, 4, 5
 * param list - a list of integers
 * returns A version of the list that is sorted in ascending order
 */
```

This function type has examples because we do care about the results. What the function type expresses is that there are multiple possible ways one could achieve this sorting, and we accept any one of them, so long as it produces correct results.

Step 2: Function Purpose Statement and Signature

In this step, you make the purpose statement and signature of a function from the wishlist more precise, based on the definitions you made in the previous step.

Purpose Statement

As the name suggests, the purpose statement describes the purpose of a function. It should contain any relevant high-level information about what the function does, and if necessary, how it does it. This is information that is entirely meant for humans. Java will ignore it.

In many cases, the purposes statement is a short sentence describing the kind of value it produces. In some cases, a purpose statement may consist of several paragraphs that describe a more complex function's behavior and its context.

For example, in the Functional Java Standard Library, the functions `Equals` and `OverlayXY` have the following purpose statements, respectively:

```
/** Returns whether two given values are equal */
```

```
/**
 * Overlays two images on top of each other, using
 * the pin position of the top image and x/y coordinates
 * relative to the bottom image's top left corner to
 * position the two relative to each other.  The
 * resulting image is sized to have space for both
 * images as they are layed out.
 */
```

Signature

The signature of a function desribes its name, its inputs, and its outputs. It has two parts: a Java signature, which is actual Java code using types that Java understands, and which is used by the *type checker*, and a Human signature, which is described in comments.

The Java Signature

The Java Signature describes the function's name and its inputs and outputs in terms of Java's type system. The general scheme is:

```
[ReturnType] [functionName]([ArgType1] [argName1], [ArgType2] [argNam
```

The curly braces `{}` belong to the function body, but should be there for Java not to get too confused. You'll add some code between them later.

Consider, for example, the following task:

```
We model traffic lights in three possible states: red, yellow,
and green.
Traffic lights switch states in the following sequence:
red -> green -> yellow -> red (and so on).
Design a function that takes a traffic light state and a number n
and returns the traffic light state after n state switches.
```

We have already designed an enumeration for traffic light states in Step 1. We'll further decide that we'll use `int` to represent the number `n`. This means that that the Java signature of this function may look as follows:

```
TrafficLightState afterSwitches(TrafficLightState originalState,
                                int numSwitches)
{}
```

The Human Signature

The human signature is there for programmers to read, and either to tell them about any information that goes beyond what Java itself can understand, or convince them that there is nothing special to worry about. It is part of the comment block above the function definition, and uses special annotations that another tool (called `javadoc`) can use. That tool is not important for now, but practicing this will come in handy later.

The human signature consists of up to three parts:

- a list of parameters/arguments (if any) and corresponding information
- if the function's return type is not void, a description of what the function returns
- if there are more complicated constraints or guarantees that a user should be aware of, a special section for those

Parameters

Each parameter documentation starts on a new line in the comment block, which is of the general form:

```
/** ...
 * @param [argName] [description]
 * ... */
```

Here, `[argName]` should match the corresponding argument name in the Java signature. The `[description]` part can span multiple comment lines, and should give a human-readable interpretation of how the argument is supposed to be interpreted. It can refer to data type definitions from Step 1, which may be particularly useful for basic data definitions.

For example, one could have:

```
/** ...
 * @param temperature The room temperature, in degrees Celsius
 *                    (>= -273)
 * ... */
```

or one could refer back to the `Temperature` basic data definition:

```
/** ...
 * @param temperature The room temperature, as a Temperature
 * ... */
```

As you can see, such descriptions may contain valuable information that goes beyond wha Java's type system can express, for example, the unit of measurement, or restrictions on th ranges of values.

Return Values

If the function's return type is not void, we need to describe more precisely what the function returns. The general form is:

```
/** ...
 * @return [description]
 * ... */
```

In contrast to parameters, the return value does not have a name. Otherwise, the description is very similar to the description of parameters, with the crucial difference that any constraints you express there are things that a user of the function can rely on and you have to guarantee, not the other way around as for parameters.

For example, we might be returning a temperature value, expressed in the Java signature as an `int`:

```
/** ...
 * @return the ambient air temperature, in degrees Celsius (>= -273)
 * ... */
```

or we use our basic data definition from above:

```
/** ...
 * @return the ambient air temperature, as a Temperature
 * ... */
```

Advanced Constraints

Lastly, you can optionally express constraints and requirements in another special part of the comment block. The general form is:

```
/** ...
 * @implSpec {Invariant: [description] |
 *           Precondition: [description] |
 *           Postcondition: [description] } ...
 */
```

That is, there may be an `@implSpec` section that contains a list of entries that each either specify an *Invariant*, *Precondition*, or *Postcondition*. We already saw *Invariant*s with record type definitions. They technically also apply to functions, but only once we get to state, so they don't play a role in functions right now. *Preconditions* express requirements that a user of the function needs to satisfy, and that you can in return rely on. We can reformulat one of the above temperature examples as follows:

```
/** ...
 * @param temperature The room temperature, in degrees Celsius
 * ...
 * @implSpec Precondition: temperature is >= -273
 */
```

Conversely, *Postconditions* express guarantees that you are making to the user of the function about its behavior, provided that the user satisfied the preconditions.

```
/** ...
 * @return the ambient air temperature, in degrees Celsius
 * @implSpec Postcondition: return value is >= -273
 */
```

You can have arbitrarily many preconditions, postconditions and invariants on a function. Just make sure that you can satisfy them, and that they make the function actually useful to anyone trying to use them. Generally, for requirements like in our examples, it makes more sense to mention them with corresponding argument or return value directly. Using the avanced constraints here is mostly useful when the conditions span multiple arguments.

Step 3: Examples

Time to come up with some examples for your function. This is a relatively simple, but hugely important step. It is important because it challenges - and hopefully solidifies - your understanding of what you need to do in the remaining steps. Coming up with concrete examples makes it easier to think through what the code you are going to write should do, and may bring up corner cases that are important to be aware of.

An example describes a particular input to the function, and the expected output that should result from that. The usual guideline is that you should come up with one example per important scenario that the function needs to handle, and at least two, wherever possible. The general form is as follows (Examples follow the purpose statement text, and go above the human signature part of the comment block):

```
/** ...
 * Examples:
 * - given: [description of inputs]
 * - expect: [description of outputs]
 * [...]
 * ... */
```

How to describe the inputs depends on the data types involved. For most basic data and enumerations, it's quite simple: you can just write down a representation of the value - where Java has literals for them, like for numbers and strings, you can directly write the literal, whereas for types like Date and DateTime you can pick a suitable representation. For other data, like records, itemizations, and some opaque types like Images, this is not as easy. In these cases, you should write a textual description of the relevant properties of the data. Try to be as precise as possible while staying readable. You will eventually need to turn these examples into code!

For example, records representing points in 2d space are simple enough to have a precise representation:

```
/** ... */
record Point(int x, int y) {}

/**
```

```
 * Calculates the euclidean distance between two points
 * Examples:
 * - given: (0, 0), (3, 4) expect: 5
 * - given: (7, -2), (2, 10) expect: 13
 * ...
 */
int distance(Point p1, Point p2) { ... }
```

Some functions are not really pure mathematical functions, and their result depends on when you call them. This particularly applies to `CurrentDate`, `CurrentTime`, and `RandomNumber`. Where these are parts of your inputs or outputs, you may state assumptions about what such a function might return:

```
/**
 * Calculates the days since a given Date
 * Examples:
 * given: [2025-01-01] expect: 53
 * (assuming the current date is [2025-02-23])
 * ... */
int daysSince(Date date) { ... }
```

A better choice may be to make your example inputs variable:

```
/**
 * Calculates the days since a given Date
 * Examples:
 * given: the date 53 days ago; expect: 53
 * ... */
int daysSince(Date date) { ... }
```

You can see many more examples of how to write examples in the documentation of the Functional Java Standard Library.

Step 4: Design Strategy

At this point, you know the purpose and signature of the function, the kinds of data that are available, and you have some examples. Now it is time to take stock of what you have and how you can use it to do what you need to do. First, this means looking at the function's arguments and their types, to see what data is available to you at this point. You may also include global constants that you may have defined.

Second, this means looking at the available functions and data type constructors, and their purpose statements/descriptions, which tell you what it is that you can immediately do with

the data that you have. This includes the basic data types and operations, such as numeric operations and `String` concatenation, as well as functions that are still on your wishlist (because you assume that they will be available eventually).

Based on this information, this step asks you to formulate a rough plan for how your function will compute its result. You generally want to keep your function small and simple, which means that if your plan becomes too complicated, you should instead come up with a new wishlist item for some helper function that will do part of the required work. When you add a new helper function to the wishlist, add it to the available functions in the wishlist, and reconsider this step (Step 4) from there.

Ultimately, you need to express your rough plan in terms of one of a selection of design strategies. In the following, we look at each of the currently available design strategies in order. With respect to the actual program code, you need to do two things:

- document your choice of design strategies below the examples section of your function's comments
- apply some changes to your function body corresponding to your choices (explained in each section below)

```
/**
 * [purpose-statement]
 * [examples]
 * Design Strategy: [your choice of design strategy] (NEW)
 * [human signature]
 */
[java-signature] {
    [modifications to body according to strategy] (NEW)
}
```

Simple Expression

A simple expression is one of:

- A variable reference (like `x`)
- A global constant (like `PI`)
- A literal (like `5`, `"Hello World"`, `false`)
- An enumeration case (like `TrafficLightState.RED`)
- A record constructor whose arguments are simple expressions (like `new Point(2,5)`)
- A primitive unary or binary operation on simple expressions (like `(speed * 3) + 5`, or `-x`)
- A lambda whose body is a simple expression (like `( x -> x + 1 )`)
- An `if` **-expression** whose components are simple expressions

As the name suggests, simple expressions reflect simple combinations of existing values or constants. If the result of your function can be computed as such, then this is the right design strategy.

If your simple expression gets a little big, you may split it up by introducing local variables, so long as those variables are again assigned with the results of simple expressions. The general scheme, and thus the modification to your body, is:

```
/** ...
 * Design Strategy: Simple Expression
 * ... */
[java-signature] {
  var ... = ...;
  return ...;
}
```

You can remove the `var ... = ...;` line in Step 5, or copy it arbitrarily many times. But you may only fill in the `...`s with either variable names (immediately after `var`s), or simple expressions. You may also add arbitrarly lines with calls to `println` before the `return` statement, provided that the argument to `println` is a simple expression.

Combining Functions

Combining functions works just like simple expressions, except that we work with extended expressions, which are like simple expressions with additional cases:

- A function call whose arguments are extended expressions (e.g. `Pow(Sin(a) / Cos(a), 2.0)`)
- A lambda call whose arguments are extended expressions (e.g. `f.apply(Exp(5.2))`)
- A record field access (e.g. `person.name()`)

To get from simple expressions to extended expressions, all conditions in simple expressions that require simple expressions components now require extended expressions as components, except for `if`-expressions, which still may only contain simple expressions - if you need to do something more complicated, you need to use the Case Distinction strategy.

If your function combination gets a little big, you may split it up by introducing local variables, so long as those variables are again assigned with the results of extended expressions. The general scheme, and thus the modification to your body, is:

```
/** ...
 * Design Strategy: Combining Functions
 * ... */
```

```
[java-signature] {
  var ... = ...;
  return ...;
}
```

You can remove the `var ... = ...;` line in Step 5, or copy it arbitrarily many times. But you may only fill in the `...`s with either variable names (immediately after `var`s), or extended expressions. You may also add arbitrarly lines with calls to `println` before the `return` statement, provided that the argument to `println` is an extended expression.

Case Distinction

So far, we know of four control-flow constructs that change what a program does based on particular values:

- `if`-expressions
- `if`-statements
- switch-expressions
- switch-statements Any case distinction that is not an `if`-**expression** whose components are simple expressions as described above is a *major* case distinction, which needs its own strategy. There should always only be at most one *major* case distinction in every function. If you need more, you need to create helper functions for that. A side-effect of this constraint is that case distinctions are incompatible with (most) Template Applications below.

Depending on which control-flow construct you use, the modification to your function bod is slightly different. In all cases, you may additionally define local variables and/or call `println`, provided the expressions used in both cases (and in filling out the holes in and around your control-flow construct) are extended expressions.

> You should generally try to use the expression form wherever possible, especially for `switch`. The `if`-expression form gets unwieldy quickly, in which case using a statement might be good for readability.

if-expression

```
/** ...
 * Design Strategy: Case Distinction
 * ... */
[java-signature] {
  ...
```

```
  return ... ( ... ? ... : ...) ...;
}
```

if-statement

```
/** ...
 * Design Strategy: Case Distinction
 * ... */
[java-signature] {
  ...
  if(...) {
    ...
    return ...;
  } [else if(...) {
    ...
  }][...] else {
    ...
    return ...;
  }
}
```

switch-expression

```
/** ...
 * Design Strategy: Case Distinction
 * ... */
[java-signature] {
  ...
  return ... switch(...) {
    case ... -> ...;
    [...]
    [default -> ...;] //where neccessary
  } ...;
}
```

For example, for our `afterSwitches` example from earlier, we may have determined that we need to both investigate the `TrafficLightState` as well as doing a case distinction on the given number. We cannot do both in the same function, so we needed to make a decision in which order to do the steps. Sometimes you have to try out one, and if things become to complicated, go back and do things the other way. In our case, the better solution is to first investigate the number, and then create a helper function to switch a traffic light once:

```
/**
 * Design Strategy: Case Distinction
 * ... */
TrafficLightState afterSwitches(TrafficLightState originalState,
                                int numSwitches)
{
  ...
  return ... switch(...) {
    case ... -> ...;
    [...]
    [default -> ...;] //where neccessary
  } ...;
}
```

switch-statement

```
/** ...
 * Design Strategy: Case Distinction
 * ... */
[java-signature] {
  ...
  switch(...) {
    case ... : {
      ...
      return ...;
    }
    [...]
    [default: {
      ...
      return ...;
    }] //where necessary
  }
}
```

Template Application

Template applications are for when you get a value of one of the more complex data types we talked about in Step 1, and want to use it, say by determining which case of an enumeration you are in, or what case and components of a general itemization you were given. This is what we wrote the templates for. To apply this strategy, you generally document both the chosen design strategy and the type whose template you are using, and then copy over the template from the data definition, remove the comments, as use the template as your new function body. You may have to adjust the variable used in the

template to a variable that is available in your function, say if you named the corresponding function argument differently.

For the new helper function we determined we needed for the `afterSwitches` function, this works as follows:

```
/** ...
 * Design Strategy: Apply Template - TrafficLightState
 * ... */
TrafficLightState cycleToNext(TrafficLightState tls) {
    ...
    return ... switch(trafficLightState) {
        case RED -> ...;
        case YELLOW -> ...;
        case GREEN -> ...;
    } ...;
}
```

Special case: While we did define templates for records and lambdas, accessing record fields and calling lambdas also counts as an extended expression, and as such you can also count using them as a function combination, and even use them in the Case Distinction strategy. Other templates, however, at not compatible with the Case Distinction strategy.

> You can only apply the template for a single value in one function. If you have multiple arguments you need to analyze, you need to define helper functions to do the additional steps, and decide on an order in which you apply the templates.

Step 5: Implementation

Time to fill in the `...`s in your function body! Having followed the Design Recipe, you are now well-prepared for this step. Refer to the Functional Java Documentation for the libraries and language constructs available to you. If there is something missing, you may have to add one or several new helper functions to your wishlist.

Finishing off the implementation of the `afterSwitches` function and its helper function (in two separate applications of the Design Recipe!) looks as follows:

```
/**
 * Design Strategy: Case Distinction
 * ... */
TrafficLightState afterSwitches(TrafficLightState originalState,
                                int numSwitches)
```

```
{
  return switch(numSwitches % 3) {
    case 0 -> originalState;
    case 1 -> cycleToNext(originalState);
    case 2 -> cycleToNext(cycleToNext(originalState));
    default -> DefaultCaseError("Invalid numSwitches argument, must b
  };
}

/** ...
 * Design Strategy: Apply Template - TrafficLightState
 * ... */
TrafficLightState cycleToNext(TrafficLightState tls) {
    return switch(trafficLightState) {
        case RED -> TrafficLightState.GREEN;
        case YELLOW -> TrafficLightState.RED;
        case GREEN -> TrafficLightState.YELLOW;
    };
}
```

Global Constants

As you implement your function, the code may involve certain constant values, most often number literals. Many of these constants are fine, particularly when they are small, can obviously be understood in terms of the code around them, and in most cases have no alternatives.

However, if you find yourself writing (and even worse, repeating) a constant that is more arbitrarily chosen, such as many of the parameters of a world program like the width and height of the world or the sizes of particular object, then we call those constants *magic constants*, which are bad. Instead of putting these constants directly into code, you should make them *global* constants, by writing something like the following outside of a function or data definition:

```
/** The width of the world, in pixels */
int WORLD_WIDTH = 500;
```

That is, you give the constant for the width of the world a globally accessible name, and use that instead of the value everywhere in your code. You also give it a comment that provides an interpretation for the value.

You can also make global constants dependent on other constants (so long as there are no circles). For example, the size of an object in a world program may determine it's maximum

coordinates:

```
/** The width of the car, in pixels */
int CAR_WIDTH = 30;
/** The minimum X-coordinate of the car, in pixels */
int MIN_CAR_X = CAR_WIDTH/2;
/** The maximum X-coordinate of the car, in pixels */
int MAX_CAR_X = WORLD_WIDTH - CAR_WIDTH/2;
```

You can see that the naming convention for global constants is to use all capital letters, with underscores separating words.

Step 6: Tests

In this step, you write testing code for your function. While it does not give you any guarantees, testing can give you a lot more confidence that your code does what it should, and it gives you some protection from accidentally breaking things when you change your code in the future, as that might make your tests fail.

Testing happens on three levels:

- at the lowest level, tests contain *assertions* about things that should be true.
- a level up, a *test* is a collection of one or more assertions.
- finally, your program's `test` function should ensure all your tests are run.

Starting in the middle, each *test* should be specified in its own function. You do not need t go through all the steps of the design recipe, but each test should have a short comment explaining what it is testing. Testing functions should follow the naming scheme `test[functionName][testName]`, have no arguments, and use `void` as the return type.

You should start by turning your examples into tests, and then potentially add some more tests. Make sure to cover all potential corner cases. Essentially, all potential branches in all called functions that are possible to reach should be executed at least once in some test, and sometimes, the combinations of certain branches in certain helper functions are important, too.

In each testing function, you need to make one or more testing *assertions*. These are functions coming from the Testing Package:

- `testEqual` asserts that two values are equal. By convention, the first argument is the *expected* value. An optional third String argument contains a message to display when the assertion fails.
- `testNotEqual` asserts that two values are not equal. An optional third String argument contains a message to display when the assertion fails.

- `testTrue` asserts that a boolean is `true`. An optional second String argument contains a message to display when the assertion fails.
- `testFalse` asserts that a boolean is `false`. An optional second String argument contains a message to display when the assertion fails. A test counts as successful if all testing assertions succeeded and nothing caused an error within the function.

Recall our `distance` function examples:

```
/** ... */
record Point(int x, int y) {}

/**
 * Calculates the euclidean distance between two points
 * Examples:
 * - given: (0, 0), (3, 4) expect: 5
 * - given: (7, -2), (2, 10) expect: 13
 * ...
 */
int distance(Point p1, Point p2) { ... }
```

For this function, the two examples would be turned into tests like so:

```
/**
 * tests Example 1 - the distance between points whose x-difference
 * is 3 and whose y-difference is 4 is 5
 */
void testDistanceExample1() {
  testEqual(5, distance(new Point(0, 0), new Point(3, 4)));
}
/**
 * tests Example 2 - the distance between points whose x-difference
 * is 5 and whose y-difference is 12 is 13
 */
void testDistanceExample2() {
  testEqual(13, distance(new Point(7, -2), new Point(2, 10)));
}
```

Finally, we need to make sure that the test functions are actually called and counted. For this, you use the function runAsTest from the testing package, which takes a reference to your particular test function as an argument.

For example, for our `distance` example above:

```
void test() {
  runAsTest(this::testDistanceExample1);
  runAsTest(this::testDistanceExample2);
  [...]
}
```

You can run the tests by either calling `test()` from your `main` function if the `main` function has nothing else to do (and then running the program), or by running the following command:

```
java --enable-preview comp1110.testing.Test [name-of-your-java-file]
```

The Design Recipe and Mutable State

Once mutable state is in the picture, there are a few adjustments to the Design Recipe, though its overall shape stays the same.

Problem Analysis and Data Design

Additionally evaluate where mutable state might play a role in the function/program that you are designing. Try to keep it at a minimum, but use it where it is useful.

In data definitions, be clear about which parts of a data definition are mutable. If you define `record` types, use the type `Cell<T>` to create a mutable field of type `T` (initialize those fields with `new Cell<>(t)`, where `t` is a value of type `T`). Make sure to take extra care around invariants for your data types, to ensure instances of your data type stay valid even in the face of mutation.

Function Purpose Statement and Signature - and Effects

The biggest change in the face of mutable state happens here. A function's interface is now not only defined by its Signature - it may also have an Effect. That is, previously, functions could only receive data via arguments, and only make data available to the outside via returning values. Now, they can also affect the mutable state of things that they can access. Java has no way of expressing or checking this, but the human-readable documentation needs to be clear about any effects that calling a function might have. Therefore, you need to be explicit about how the function affects the mutable state of any data or variable in the program outside of its local variables. The following part is added to the comment block above a function:

```
Effects: [describe the effects of the function here]
```

This part should appear between the `Design Strategy` part (which will appear above it) and the "human signature" part (`@param/@return`). At this point in the design recipe, it will be inserted between the purpose statement and the human signature.

Examples

Examples now not only need to mention the inputs and corresponding expected outputs to the fuction, but also describe the relevant parts of mutable state before the start of executing the function, and the relevant parts of mutable state after the function returns. For example:

```
int counter = 0; //generally bad to use mutable global variables,
                 //just used as a short example
/**
 * Increases the global counter and returns its new value
 * Examples:
 * - Given: [no inputs] - the global counter is 4
 *   Expect: [no outputs] - the global counter is 5
 * - Given: [no inputs] - the global counter is 16
 *   Expect: [no outputs] - the global counter is 17
 * Design Strategy: Simple Expression
 * Effects: the global counter is increased by one
 */
int incCounter() {
    counter = counter + 1;
    return counter;
}
```

Design Strategy / Implementation

No particular changes here, except that you can use assignment statements and other stateful constructs in the implementation step.

Tests

Testing code that mutates state is a bit tricker than testing purely functional code. This is because the behavior of functions is now not only determined by their arguments, but also by mutable state. You essentially have two options for dealing with this, explained below. An additional factor is that you may not only need to test the return values of functions against expections, but also parts of the global state.

Explicitly Setting Up State

Before every test, you can try to explicitly set up the relevant state so that you know exactly what your function will do. For example:

```
counter = 6;
testEqual(7, incCounter());
testEqual(7, counter);
```

Checking The Current State

Before every test, you can also try to retrieve the relevant state, and calculate the expected results based on it. For examples:

```
int currentCounter = counter;
testEqual(currentCounter+1, incCounter());
testEqual(currentCounter+1, counter);
```

Where to Apply The Design Recipe

You should apply the design recipe and go through the appropriate steps for all functions that you write for an assignment, with the following exceptions:

- the functions `main` and `test`
- all of your testing functions, which have their own small comment described in Step 6
- any functions that are only prescribed as part of the test interface for an assignment. provided that you do not use the function yourself. In some cases, you can reuse a function that you wrote as part of the testing interface, in which case that function still needs to follow the steps of the design recipe. Otherwise, simply put a comment on top of the function: `// testing interface`.
- you do not need to write tests for functions whose return type is `Image`

The Design Recipe and World Programs

World programs mostly follow the design recipe normally - you just start with a wishlist for the `draw` / `step` /etc. functions. The key adjustment is that you do not have to write tests for images, but you should still write up examples that describe what should happen.

Acknowledgement of Country

The Australian National University acknowledges, celebrates and pays our respects to the Ngunnawal and Ngambri people of the Canberra region and to all First Nations Australians on whose traditional lands we meet and work, and whose cultures are among the oldest continuing cultures in human history.