Exercise 3: Name and Type Analysis

Representing Types

```
Type =          // a type is either void or a proper type
   | VoidType
   | ProperType

ProperType =    // a proper type is eiher a reference type or a value type
   | ReferenceType
   | ValueType

ValueType =     // there are two value types (int and boolean)
  | IntType
  | BooleanType

ReferenceType =
  | ArrayType // a reference type is either an array type
  | NullType  // or null

ArrayType = // an array type has an element type (of proper type)
  {elementType}
```

As seen above, we have an abstract superclass Type. The only thing all types have in common is a name. Therefore, **Type** offers `getName()` which must be implemented by all types. **Type** does not contain the name as field since some types return a constant name (e.g. *void*). Furthermore, all types must implement `isSubtypeOf()` which is used to check for type compatibility.

Each type is represented by exactly one object, to allow comparison via `==`. The following types are represented by singletons accessible via `<typename>.INSTANCE`:

- VoidType
- IntType
- BooleanType
- NullType

For each class in the source program one object of **ReferenceType** is instantiated. **The original here discusses the class types . . .**

Arrays are a special case of a reference type. Each instance of **ArrayType** has an element type of **ProperType**. The subtype relation is covariant to the element types except for two special cases: **The original here discusses the usage of a super type for references; this is different for our portfolio!**

Analysis

```
MJFrontend frontend = new MJFrontend();
MJProgram program = frontend.parseString(input);
// [..]
Analysis analysis = new Analysis(program);
analysis.check();
```

We have a class Analysis that takes the abstract syntax tree and performs analysis in the function `check()`. Afterwards a list of errors can be retrieved by `analysis.getTypeErrors()`. Analysing works in the following steps:

1. First, objects for each type in the program are instantiated.
2. Next, we check for signature compatibility of overwritten methods using **`isSubTypeOf()`** on the return types and the parameters.
3. Finally, we check names and types in the code of each method. For this step, we set up a symbol table with . . . . **The original goes into detail regarding the symbol table**. Then, we walk through the method's body using the AST's visitors.

```
// overall pseudo code for Analysis.check()
instantiateAllTypes()
checkMethodSignatures()
for class in classes do
  for method in class.methods do
    recursivelyCheck(method.body)
```

Symbol Table

The symbol table is implemented efficiently using a map from identifiers to stacks of declarations and a stack of scopes. In both cases we use `ArrayDeque<>` for the stacks. We differ between local declarations and field declarations to correctly determine which name can be reused (see package `analysis.names`).

Main Method

Since the main class and the main method are a special case in the grammar they are not checked by our design above. Therefore, we need to check them separately with `Analysis.checkMain()`. **The original here discusses the handling of the main class; this is different for our portfolio!**