Semantics of the Brown Decaf Programming Language[*]

Steve Reiss
CS 126 Spring 2006

1 Introduction

This document describes the semantics of the programming language Decaf. Decaf is a subset of Java, and hence, most Decaf statements have the same semantics as if they were written in Java. The primary difference between the two languages is in the statements allowed. The syntax specification eliminates a lot of Java features, such as threads, exceptions, floating-point numbers, etc. This document specifies which syntactically valid Decaf programs are also semantically valid (i.e., should compile without errors), and what the effect of executing semantically valid programs is. The topics covered in this document are:

- Summary of Features — what is and isn't a Decaf program
- Semantic Checking — how to determine whether a Decaf program is semantically sound
- Evaluation — what should happen when you execute a Decaf program

Please report any bugs, ambiguities, or other problems with this document to `geddon@cs.brown.edu` and `spr@cs.brown.edu`.

2 Object-Orientation and Terminology

Decaf and Java programs are object-oriented. In both languages, a program consists of a collection of *classes.* A class defines two things: a set of *fields*

[*]Based on the document *Semantics of the Decaf Programming Language*, by Dan DuVarney and Purush Iyer at the North Carolina State University with help from Manos Renieris at Brown.

(also called *data members*) that form a template for creating *objects* (also called *instances* of the class), and a set of operations that can be performed on objects of the class (called *methods* or *member functions*). The term *member* of a class `C` refers to any method or field of `C`.

A class `X` may be declared to `extend` class `Y`, in which case we say that `X` is a *subclass* of `Y` (equivalently, we may say: `X` *inherits* from `Y`, `Y` is a *superclass* of `X`, or `X` is *derived* from `Y`). If `X` is a subclass of `Y`, then `X` *inherits* all the fields and methods defined in `Y` and defines new fields and methods of its own. If `X` defines a field `f` with the same name as a field in `Y`, then `X`'s `f` is said to *hide* `Y`'s `f`. If `X` defines a method `m` with the same name as a method in `Y`, then `X`'s `m` is said to *override* `Y`'s `m`. For example:

```
class Y {
  public int f() { return 2; }
  public int g() { return 3; }
}

class X extends Y {
  public int f() { return 4; }
  public int h() { return 5; }
}
```

In this program, `X` has three methods: `f`, `g`, and `h`. The method `f` overrides the `f` supplied by `Y`. the method `g` is inherited from `Y`. The method `h` is defined in `X`.

A primary feature of inheritance is that is makes Class `X` into a *subtype*[†] of class `Y`, so that any method that expects an object of class `Y` can be passed an object of class `X`. Subtyping works in combination with *dynamic binding.* Dynamic binding is the primary feature that distinguishes object-oriented programs from those of other paradigms. When a method invocation is dynamically bound, the method to be called is selected at runtime. For example:

```
int
do_f(Y y)
{
```

[†]See section 6.1.1 for the definition of a subtype in Decaf

```
  return y.f();
}
```

You might expect that the function `do_f` would always return the value `2`, since `y` is an instance of class `Y`, and `Y`'s `f` always returns 2. However, that's not the case. Consider the following code snippet:

```
  X x;
  return do_f(x);
```

In this case, an object of class `X` is passed to `do_f`. Class `X` overrides `Y`'s definition of `f`. When `y.f()` is called, the method is selected dynamically by looking in the original class of the object that is bound to the variable `y`. We will use the term *dynamic type* to refer to the original class of an object (the class whose constructor created the object). The *static type* of `y` is the class that `y` is declared to have (`Y`, in this case).

The dynamic class of `y` is `X`, so the dynamic binding of `y.f` chooses the `f` defined in class `X`. `X`'s `f` is called instead of `Y`'s `f`, resulting in a value of `4`.

3 Decaf Features

The following features are expressible in the Decaf grammar, but aren't part of the language:

1. Static fields.
   Methods can be `static`, but not fields.

2. Dynamic allocation of arrays of objects.
   The `new` operator need only work with arrays whose base type is `int`, `char`, or `bool`. Multidimensional arrays are allowed as long as the base type is one of the above. It should also be possible to declare a variable whose type is an array of some class type (although it wouldn't be possible to initialize it).

3. Overloading.
   A Decaf program can never define two methods of the same name in the same class.

3.1 Supported Features

The Decaf language includes the following features:

- **Literals** of type `int`, `boolean`, `Object`, `String`, and `char`.
- **Unary integer operators** `+` `-`
- **Unary boolean operator** `!`
- **Binary integer operators** `+` `-` `*` `/` `%`
- **Binary boolean operators** `&&` `||`
- **Polymorphic assignment operator** `=`
- **Polymorphic equality operators** `==` `!=`
- **Integer relational operators** `<` `<=` `>=` `>`
- **Arrays** that define uniform collections of elements (all of the same type), and are allocated via the `new` operator.
- **Array Element Lookup** using expressions of the form A`[`i`]` to fetch the ith element of array A.
- **Classes** that synthesize a type from a set of fields and methods.
- **Objects** that are instances of a class and are allocated via the `new` operator. The class of an object defines the set of methods applicable to the object, and the fields stored in the object.
- **Fields** that are "variables stored in an object." Each time a new object is created, it is allocated its own copy of all the fields in the object.
- **Methods** that are functions defined for a particular class.
- **Constructors** that are special methods with the same name as the class and no return type. The constructor is called every time a new object is created. The purpose of a constructor is to initialize the fields of the object.

- **Dynamic method lookup** using the dot (`.`) operator. The left hand side of the dot expression must evaluate to an object, and the right hand side must be the name of a method.
- **Static method lookup** using expressions of the form $ClassName\texttt{.}MethodName$.
- **Field Lookup** using the dot operator. The left hand side of the dot expression must evaluate to an object, and the right hand side must be the name of a field.
- **Method invocation** by passing arguments to a method (similar to a function call).
- **Expressions** combining all of the above features with precedence rules[‡] and the ability to use parentheses to override precedence.
- **While loops** that repeatedly execute a statement as long as a given boolean expression remains true.
- **Break statements** that force an exit from a while loop.
- **Continue statements** that force a while loop to start over from scratch.
- **If statements** that conditionally execute a statement when a given boolean expression holds true. The if-statement may optionally have an else statement that is executed only if the condition holds false.
- **Block statements** that group a sequence of statements together.
- **Return statements** that terminate execution of a method and return a value back to the spot from which the method was invoked.

4 Declaration Processing

Each declaration of a class, field, method, or variable takes a name and associates it with a type. Decaf programs may then refer to the declared items by giving the name. Your compiler will generate a *symbol* for each declaration. The symbols should be organized into symbol tables. The purpose of

[‡]The precedence rules are detailed in Syntax of the Decaf Programming Language

a symbol table is to make it easy to find the type (as well as other information) associated with a name, paying particular attention to the *scope* of each name. The scope of a name is the region of the program where a particular name is visible.

Decaf allows expressions and types to contain forward references to classes, fields, and methods that are declared later in the source file. Hence, compilation of a Decaf program is best done in (at least) two passes:

1. Parse the input file, process all declarations, and build abstract syntax trees and symbol tables.

2. Resolve all references to symbols, check for type mismatches and other semantic errors, and generate code.

One possible way to organize the symbol tables is as follows. First, place all the class symbols in a *class symbol table*. Second, each class symbol should contain two symbol tables — a *method symbol table*, and a *field symbol table*. A method symbol table will contain all the method symbols for a particular class, and a field symbol table will contain all the field symbols for a particular class. Third, each method symbol should contain a *variable symbol table*. A variable symbol table will contain symbols for all the formal parameters and other local variables of a particular method.

4.1 Class Declarations

The top-level of a Decaf program is a sequence of one or more class declarations. Within each class declaration appear zero or more field and method declarations. For example, consider:

```
class IntNode {
  private int x;
  private IntNode next;
  public int value() { return x; }
  public IntNode next() { return next; }
  public IntNode(int xi, IntNode nexti) { x = xi; next = nexti; }
}
```

This declares the class `IntNode` to have two fields (`x` and `next`), two methods (`value` and `next`), and a constructor.

4.1.1 Superclasses

The `extends` keyword is used to declare one class to be superclass of another. For example:

```
class Point {
  public int x;
  public int y;
}

class ColorPoint extends Point {
  public int color;
}
```

In this example, the class `Point` is a superclass of `ColorPoint` (and, equivalently, `ColorPoint` is a subclass of `Point`). Objects of class `ColorPoint` will have three fields — `color`, `x`, and `y`. The fields `x` and `y` are inherited ¿from the superclass `Point`.

In Decaf, Every class has a superclass except the built-in class `Object`. If a class is declared without the `extends` keyword, then by default it has the superclass `Object`. So, in the previous example, `Point` is a subclass of `Object`.

Decaf also requires that every class be declared after its superclass. The purpose of this requirement is to prevent a "cycle" of superclasses.

4.1.2 Built-in Classes

The Decaf runtime system includes the following classes. These classes should be "pre-declared" before the compiler starts processing any source file.

```
class Object
{
  public Object();
}

class String extends Object
{
  public String();
}

class IO extends Object
{
  private IO();
  public static void putChar(char);
  public static void putInt(int);
  public static void putString(String);
  public static int  peek();
  public static int  getChar();
  public static int  getInt();
  public static String getLine();
}
```

Note that no code needs to be generated for these classes. You will be given a runtime library that includes the `IO` class. Your compiler needs to generate symbols for the three built-in classes and place them in the symbol table before beginning compilation of the user's input file.

4.1.3 Class Declaration Visibility

All class declarations are globally visible (and forward references to yet-to-be-defined classes are allowed). No two classes can have the same name. Declaring two classes with the same name is an error. This means that input files cannot define classes named `Object`, `String`, or `IO`.

4.2 Field Declarations

A field is a variable that is allocated once per object. The declaration of a field gives the access modifier, the type, and then the name of the field. For example, consider:

```
class Y {
  public int[] members;
  private X myX;
}
```

In this example, class `Y` has two fields: a publicly accessible field `members` whose type is array of integers, and a private field `myX` whose type is class

`X`. Note that it's legal to use `X` as the type of a declaration before class `X` is defined.

Field declarations are subject to the following restrictions:

- The modifier `static` cannot be used.
- The field should not have an initializer (instead, the constructor should initialize it).
- Each class can define only one field with a given name (field names must be unique within the class). It is legal for a class to define a field with the same name as field in its superclass — in which case, the subclass hides the superclass' field (although both fields are still stored in all instances of the subclass).
- Only one access modifier (`public`, `private`, or `protected`) should be used.
- If no access modifier is given, then `public` is assumed.

4.3 Local Variable Declarations

Local variables are variables that belong to a particular method. Declarations are similar in syntax to field declarations, with the exception that there may be an initial value given for the local variable (as long as the variable isn't a formal parameter). Each local variable has a type and name. For example:

```
int fac(int n)
{
  int i = 2;
  int f = 1;

  while (i <= n) {
    f = f * i;
    i = i + 1;
  }
  return f;
}
```

In this example, the method `fac` has three local variables. The first is the formal parameter `n`. The other two are `i` and `f`. The type of all three declarations is `int`.

4.4 Method Declarations

Methods are functions belonging to a class. A method declaration is similar to a function definition in a non-object-oriented language: first appear access and storage modifiers, second a return type, third the the name of the argument, fourth the formal parameter declarations, and finally the body of the method. For example:

```
class Car {
  private int my_displacement;
  private String model;
  int displacement() { return my_displacement; }
  String model() { return myModel; }
  void setDisplacement(int d) { my_displacement = d; }
  Car(int d, String mod) { my_displacement = d; myModel = mod; }
}
```

Class `Car` contains four methods:

- `displacement` that has no formal parameters and returns an `int`.
- `model`, that has no formal parameters and returns a `String`.
- `setDisplacement` that has one formal parameter of type `int` and a return type of `void`.
- `Car`, a constructor that has two formal parameters — the first parameter has type `int`, and the second has type `String`.

Method declarations are subject to the following restrictions:

- Each class can define only one method with the same name. It is legal for a class to define a method with the same name as a method in its superclass — in which case, the subclass overrides the superclass' method.

- Only one access modifier (`public`, `private`, or `protected`) should be used.
- If no access modifier is given, then `public` is assumed.
- If class `X` overrides class `Y`'s non-`static` method `m`, then `X`'s m and `Y`'s `m` must have the same argument and return types.

4.4.1 Constructors

Every class has a constructor. The constructor is a special method with the same name as the class, and no return type. Unlike Java, Decaf doesn't allow overloading, so there is only one constructor per class. If a class is defined without a constructor, then it receives the following default constructor, that takes no arguments, and then calls the superclass constructor with no arguments:

```
// default constructor for
//      class X extends Y
X()
{
  super();
}
```

Calls to the superclass constructor are only allowed as the first statement of a constructor. If a constructor is defined without a call to the superclass constructor, then there is a default superclass constructor call passing no arguments (note: this will cause an error if the superclass constructor expects arguments). For example:

```
class Y {
  int y;
  Y(int n) { y = n; }
}

class X extends Y{
  int x;
  X() { x = 2; }
}
```

When the default superclass constructor calls are added, we end up with the following:

```
Y(int n) { super(); y = n; }
X() { super(); x = 2; }
```

The constructor for `Y` is correct because the constructor for `Object` expects no arguments (see section 4.1.2 for details). The constructor for `X` has a semantic error because it doesn't pass enough arguments to `Y`'s constructor.

5 Resolving Names

Once all the declarations in a Decaf program have been processed and the symbol table has been constructed, a second pass is required to resolve all expressions and types that contain names of classes, fields, methods, and local variables (A two-pass approach is required because of forward references).

The first step in resolving a name `x` is to determine from context what kind of symbol a name refers to.

(Case 1)	If `x` appears in a type description, then `x` is the name of a class.
(Case 2)	If `x` appears by itself as the left hand side of a method call (e.g., `x(2)`), then `x` must be the name of a method.
(Case 3)	If `x` appears as the right hand side of a dot expression on the left hand side of a method call (e.g. `g().x(2)`), then `x` must be the name of a method.
(Case 4)	If `x` appears as the right hand side of a dot expression **not** on the left hand side of a method call (e.g. `g().x - 1`), then `x` must be the name of a field.
(Case 5)	If `x` appears on the left hand side of a dot expression (e.g., `x.y`), then `x` is the name of a local variable, field, or, class.
(Case 6)	If `x` doesn't fit any of the previous three cases, then `x` is the name of a local variable or field.

We cover each of these cases in the next four subsections..

5.1 Resolving Class Names (Case 1)

To resolve (Case 1) names, consult the symbol table. Since all classes are "globals" and have a unique name, this should be easy. It's a semantic error if a type contains the name of an undefined class. For example:

```
  X[] members;
```

In this example, the type `X[]` contains the name `X`. Since `X` appears in a type description, we know by (Case 1) that `X` is the name of a class.

5.2 Resolving Method Names (Case 2)

To resolve (Case 2) names, start by looking in the class containing the expression for a method with the given name. If no method is defined, in that class, look in its superclass. Keep following the superclass chain. If the method isn't eventually defined by one of the superclasses, the name is undefined and a compilation error results. For example:

```
class Y {
  int f() { return 100; }
}

class X extends Y {
  int g() {
    return 10 + f() + g() + h();
  }
}
```

During resolution of the expression `10 + f() + g() + h()`, three names must be resolved. First, the name `f` is resolved. The search starts in class `X`. `f` isn't defined in `X`, so the search proceeds to `Y`, where `f` is defined. Second, the name `g` is resolved. The search starts in class `X`, where `g` is defined, so the search is concluded successfully. Finally, the name `h` is resolved. The search starts in class `X`, proceeds to class `Y`, and then to class `Object` (the superclass of `Y`). `h` isn't defined in `Object` and `Object` has no superclass, so the search fails. The name `h` is undefined and compilation fails with a semantic error.

5.3 Resolving Method Names (Case 3)

To resolve (Case 3) names, you must first determine the class in which to look for the method. The class is determined by the static type of the left hand side of the dot expression. The type of the left hand side must be a class type (otherwise a type mismatch occurs — this will be covered in detail in section 6). The search for the method then starts in that class and proceeds just like (Case 2). For example:

```
class Y {
  int f() { return 100; }
}

class X extends Y {
  int g()
  {
    Y y = new Y;
    return y.f();  // y.f needs to be resolved
  }
}
```

The expression `y.f()` in method `g` contains an unresolved method name `f`. `f` is resolved by finding the type of `y`, that is the class `Y`, in this case. The search for `f` starts in class `Y` and succeeds since `Y` contains a method `f`.

5.4 Resolving Field Names (Case 4)

(Case 4) names are resolved in manner very similar to (Case 3). The expression has the form e`.f`. The class search for `f` is the static type of the expression e. `f` must be a field defined in the class of e, or inherited from a superclass of the class of e. For example:

```
class Y {
  int f;
}

class X extends Y {
  int g()
  {
    Y y = new Y;
    ...
    return y.f;  // y.f needs to be resolved
  }
}
```

The expression `y.f = 5` in the return statement of method `g` contains an unresolved field name `f`. This is resolved by computing the static class of `y` (class `Y`, in this case), and then finding the `f` defined in class `Y`.

5.5 Resolving Variable, Field, or Class Names (Case 5)

The rules for resolving a (Case 5) name require a three-phase search. First, local variables are checked. If a local variable has the name being looked-for, then that variable is the result of the search. Second, if no local variable matches, then the class containing the expression being resolved is checked for fields with the name. If a field is found, then the name refers to that field. If no field matches, then the superclass is checked, continuing until there is no superclass. Third, if the first two searches fail, then the name must refer to a class, and must be part of an expression accessing a `static` member of that class. Consider the following example:

```
class Y {
  int x;
  z w;
}

class X extends Y {
  int f(X x) { return x.y + w.g() + z.g(); }
}

class z {
  static int g() { return 5; }
}
```

In this example, the expression `x.x + w.g() + z.g()` contains three (Case 5) names: `x`, `w`, and `z`. First, `x` refers to a local variable (the formal parameter to function `f`). Second, there is no local variable named `w`, so the search for `w` progresses to the second stage. Class `X` is checked for a field named `w` (there is none). The search then moves on to class `Y`, where `w` is a field. Hence, the name `w` refers to the field `w` defined in class `Y`. Finally, `z` is resolved. There is no local variable and no field named `z`, so `z` must refer to a class.

5.6 Resolving Variable or Field Names (Case 6)

The rules for resolving a (Case 6) name are quite similar to (Case 5). The only difference is that the name cannot be a class name, so the third phase of the search is skipped. For example:

```
class Y {
  int y;
}

class X extends Y {
  int x;
  int f(int w) { return x + y + z; }
}

class z {
 ...
}
```

In this example, the expression `x + y + z` has three names to be resolved. The name `x` refers to the field `x` defined in class `X`, the name `y` refers to the field `y` defined in class `Y`, and the name `z` cannot be resolved (even though there is a class `z`), resulting in a compilation error.

5.7 Hidden Fields

When a subclass declares a field with the same name as a field in the superclass, the subclass field *hides* the superclass field. For example:

```
class Y {
  int y;

  int g() { return y; }
}

class X extends Y {
  boolean y;

  int f() { return y; }
}
```

In this example, the method `f` returns the field `y` of class `X`, while the method `g` returns the field `y` of class `Y`. This is determined at compile time. No dynamic binding of fields ever occurs. Both `y` fields are stored in objects of class `X`, so that a memory diagram of an object of class `X` might look like this:

Name	**Value**
y	102
y	*true*

Which `y` field is returned by expression of the form *e*`.y` depends on the static type of *e*, as illustrated in the previous example.

5.8 Nesting and Local Variables

A final complication to resolution of names is that a local variable is only visible from the point where it is declared until the end of the block in which the declaration occurs. This is illustrated by the following example:

```
class X {
  int z;

  int f(int z)
   {
     if (z > 0) {
        int z = x * 2;

        return z;  // line 1
     }
     return z;  // line 2
   }
}
```

The `z` returned by `// line 1` refers to the `z` defined inside the `if` statement. The `z` returned by `// line 2` refers to the formal parameter `z` of function `f`. Neither `z` refers to the field `z` of class `X`.

5.9 Resolving `super`

`super` can be used in non-`static` methods to access fields and methods defined in the superclass. If `super.x` is used in a method invocation, the method to call is bound statically at compile time to the method `x` in the superclass.

```
class Y {
  int f() { return 2; }
}

class X extends Y {
  int f() { return 3; }
  int g() { return super.f(); }
}
```

The result of `g()` will be 2, not 3.

`super` cannot be used in `static` methods.

6 Semantic Checking

The purpose of semantic checking is twofold. First and foremost, programs with semantic errors should be rejected. For example, consider the expression:

```
  x = "eat" + 2;
```

This expression contains a *type mismatch* — the attempt to combing string and integer values by applying the `+` operator. An expression that contains no type mismatches is called *type-safe*.

The second job of the semantic checker is to collect the type information because it will be needed to generate code. This section will give a mostly informal specification of the Decaf type system. The specification will consist of:

- what types are
- which types are valid
- how to compute the type of a declaration
- which expressions are type-safe

- what the type of an expression is
- which statements are type-safe
- semantic checks to be done in addition to type checking

It is the job of the type checker to ensure that all declared fields, methods and local variables have valid types, and that all expressions and statements are type-safe.

6.1 Valid Types

There are four kinds of types in Decaf:

- *primitive types* (`int`, `char`, `boolean`, `void`)
- *class types*
- *array types*
- *function types* (that map from a tuple of n types to a result type)

We will use the following notation to describe types:

(Primitive)	`int`, `char`, `boolean`, `void` are types
(Class)	If X is a valid identifier then $C(X)$ is a type
(Array)	if T is a type other than `void` or a function type then $A(T)$ is a type
(Function)	If $T_1, T_2, \ldots T_n$ are types other than `void` or function types (note: n can be 0), and T' is not a function type then $(T_1 \times T_2 \times \ldots \times T_n) \rightarrow T'$ is a type

6.1.1 Subtypes

An important relation between types is the *subtype* relation. Basically, T_1 is a subtype of T_2 if T_1 can represent every value that T_2 can represent, and all the operations that work with T_2 will also work with T_1. The subtype relation is defined as follows:

- Every type is a subtype of itself.

- `int` is a subtype of `char`, and `char` is a subtype of `int`.
- For all classes X and Y, if X is a subclass of Y then $C(X)$ is a subtype of $C(Y)$.
- If T_1 is a subtype of T_2, then $A(T_1)$ is a subtype of $A(T_2)$.
- If, for $i = 1 \ldots n$, T_i is a subtype of T'_i, and T'_r is a subtype of T_r, then $(T'_1 \times T'_2 \times' \ldots \times T'_n) \to T'_r$ is a subtype of $(T_1 \times T_2 \times \ldots \times T_n) \to T_r$.

Note that the subtype relation "reverses" itself when it comes to function arguments. This property is called *contravariance.* Also notice that `char` is not a subtype of `int`, so $A($`char`$)$ is not a subtype of $A($`int`$)$, and so on.

6.2 Checking Declarations

The Decaf grammar makes it impossible for declarations to contain invalid types, with a couple of minor exceptions:

- `void` can only be used as the return type of a function
- All class names must refer to defined classes.

The type resulting from a declaration is relatively straightforward:

- A primitive type appearing in a type declaration represents itself
- An identifier X appearing in a type declaration represents type $C(X)$
- A type declaration T followed by `[]` represents type $A(T)$ (`[][]` represents $A(A(T))$, and so on)
- A method declared to have argument types $T_1 \ldots T_n$ and return type T' has type $(T_1 \times \ldots \times T_n) \to T'$

There are a few other semantic checks that should be done on declarations. Some of these were given in section 4. Here is a complete list:

- No two classes can have the same name
- Every subclass must be declared after its superclass

- No class can declare two methods with the same name (a class can declare a method with the same name as an inherited method)
- No class can declare two fields with the same name (a class can declare a field with the same name as an inherited field)
- `static` can only be used with methods, not fields or local variables
- No declaration should have more than one access modifier (`public`, `private`, or `protected`)
- `void` can only be used as the return type of a method
- If a method `m` is declared `static` in class `X`, then any redefinition of `m` in a descendent class must also be `static`.
- If a method `m` is declared non-`static` in class `X`, then any redefinition of `m` in a descendent class must also be non-`static`.
- If a non-`static` method `m` in class `X` has the same name as a method `m` in a superclass `Y`, then the (function) type of `X`'s `m` must be the same as the type of `Y`'s `m`.

6.3 Checking Expressions

One of the jobs of the type checker is to make sure that for each expression, all the subexpressions have an appropriate type. To type-check an expression e, the type-checker will need to start out at the leaves of the abstract syntax tree for e, and propagate type information upwards. At each node in the syntax tree, the type checker will need to verify that the types of the subexpressions are correct. The set of type-safe expressions and their types are specified as a system of rules in the next sections.

6.3.1 Checking Literals

All literals are inherently type-safe. The rules for literals are:

- The type of the literal `null` is $C($`Object`$)$
- The type of string literals is $C($`String`$)$

- the type of character literals is `char`
- the type of integer literals is `int`
- the type of `true` and `false` is `boolean`

6.3.2 Checking Unary Expressions

The unary operators `+` and `-` work only on `int` (or subtypes of `int`) subexpressions and always return an `int` result. For example, `-'c'` is a type-safe expression, while `-"cat"` is not.

The unary operator `!` works only with a `boolean` subexpression and returns a `boolean` result. For example, `!false` is a type-safe expression, while `!0` is not.

6.3.3 Checking Identifiers

The type of a field, method, or variable identifier appearing in an expression is determined by looking up the name using the rules in section 5, and returning the type stored in the symbol table. An identifier appearing by itself is always type-safe. If the identifier cannot be resolved using the rules in section 5, then a compilation error occurs.

6.3.4 Checking Binary Expressions

The binary mathematical operators `+`, `-`, `*`, `/`, and `%` only work on integer (or subtypes of integer) subexpressions and always return integer results. For example, `3 * 4` is type-safe because both subexpressions have type `int`, while `"hello" * 2` contains a type mismatch.

The binary operators `&&` and `||` only work on `boolean` subexpressions and always return `boolean` results.

The equality operators `==` and `!=` work on on any two subexpressions whose types are related by the subtype relation (in either direction). The result is always `boolean`. For example, `null == "xxx"` is type-safe because the type of `null` is C(`Object`), the type of `"xxx"` is C(`String`), and C(`String`) is a subclass of C(`Object`).

The other relational operators (`<`, `<=`, `>=`, `>`), only work with `int` subexpressions, and always return a `boolean` result. For example, `"cat" < "dog"` is a type mismatch because the subexpressions aren't of type `int`.

The assignment operator `=` requires the following:

- the left subexpression must be an l-value. An l-value is a name, a dot-expression, or an array index operation, but not a function call, unary expression, binary expression, or `new` operation.
- the type of the right subexpression must be a subtype of the left subexpression.

For example, if `x` is an object of class `X`, `y` is an object of class `Y`, `X` is a subclass of `Y`, then `x = y` is type-safe but `y = x` is not. `(x + 1) = 2` is not type-safe because `(x+ 1)` isn't an l-value.

6.3.5 Checking `this` and `super`

The type of `this` is the class in which the `this` occurs. The type of `super` is the superclass of the class in which the `super` occurs. Both `this` and `super` can only be used in non-`static` methods. Consider:

```
class X {
  int f() { return 0; }
}

class Y extends X {
  int f() { return 1; }
  int g() { return this.f() + super.f(); }
}
```

In the body of method `g`, the type of `this` is C(`Y`) and the type of `super` is C(`X`).

6.3.6 Checking Parentheses

Parentheses have no effect on the type or correctness of an expression. For example, `((("cat")))` has type C(`String`).

6.3.7 Checking Method Calls

In order to be type-safe, a method call of the form e`(`e_1`, `e_2`, `$\ldots e_n$`)`, must satisfy the following requirements:

- The type of the subexpression e must be a function type $(T_1 \times T_2 \times \ldots T_m) \rightarrow T'$ (the type of e can be determined by resolving e using the rules in section 5.2).
- The number of formal parameters must match the number of actual arguments (n must equal m).
- The type of each actual argument must be a subtype of the corresponding formal argument type.

The resulting type is the return type of the method (T'). Consider the following methods:

```
int f(int x) { ... }
boolean g(int x, boolean y) { ... }
```

Here are some expressions and the results of type-checking:

`f(2)`	is type-safe and has type `int`
`g(2)`	is not type-safe because the number of arguments is wrong.
`g(2, 2)`	is not type-safe because the second argument has type `int`, that is not a subtype of `boolean`.
`g(4, true)`	is type-safe and has type `boolean`.

There are two additional requirements:

- `static` methods cannot contain calls to non-`static` methods.
- Constructors may not be called except within `new` expressions and as the first statement of a constructor.

6.3.8 Checking `new`

There are two distinct cases of `new`. One case is when `new` is used to allocate a new object. In that case, `new` operation is type-checked like a method call to the constructor. The number of actual arguments must equal the number of formal parameters in the constructor, and the actual argument types must be subtypes of the arguments expected by the constructor. For example:

```
Class X {
  X(int x, boolean b) { ... }
}
```

Here is are some expressions and the results of type-checking:

`new X(1)`	is not type-safe because the number of arguments is wrong.
`new X(2, 2)`	is not type-safe because the second argument has type `int`, that is not a subtype of type `boolean`.
`new X(2, true)`	is type-safe and has type $C($`X`$)$.

The second case is when `new` is used to allocate an array. In that case, there must be one or more actual arguments, and the actual arguments must all be a subtype of `int`. The base type of the array must be a primitive type (remember our restriction from section 3). Here are some examples:

`new X[3][4]`	is not legal (the base type isn't primitive)
`new int[3][4]`	is legal with type $A(A($`int`$))$
`new int[3][true]`	is not legal because the second argument is not type `int`

6.3.9 Checking Array Indexing

To check an expression of the form e_1`[`e_2`]`, the following requirements must be satisfied:

- subexpression e_1 must have an array type
- subexpression e_2 must have type `int`

To get the result type, peel the outermost A from the type of e_1. For example, if `x` is declared as

```
int[][] x = new int[][](10, 10);
```

then type of `x` is $A(A($`int`$))$. In this context, the expression `x[5]` is type-safe and has type $A($`int`$)$.

6.3.10 Checking Dot Expressions

To check an expression of the form e_1`.`x, first check e_1. Either (1) e_1 is a type-safe expression and e_1's type is be a class type of some sort, or (2) e_1 is the name of a class (the rules for determining this are given in section 5).

For either case (1) or (2), resolve `x` using the rules of section 5. If `x` turns out to be the name of a field, then the expression is type-safe and the type of the expression is the type of the field. If `x` resolves to a method, the expression is type-safe and the type of the expression is the type of the method. A compilation error occurs if `x` cannot be resolved.

There are some additional semantic checks to be performed. First, for expressions of form (2), `x` must refer to a `static` method. Second, `x` must be *accessible*. The accessibility of a member of class X is determined by consulting the following table:

Declared Access	**Where Visible**
`public`	Everywhere
`protected`	Only within class X and subclasses of X
`private`	Only within class X

Attempting to access a member outside of the region where it is visible causes a compilation error.

Finally, there is a special dot expression for array types. e`.length` computes the length of an array. The type of e must be an array type. The resulting type of the entire expression is `int`. For example, if `a` is declared as `int[][] a`, then `a.length` is a type-safe expression of type `int`.

6.3.11 Checking Names

Checking a name within an expression is a two-step process: First, the name must be resolved using the rules of section 5. Once a match is found, there are two additional hurdles that must be cleared:

1. If the name matches a class member, the member must be accessible (see the table in section 6.3.10)

2. If the name appears within a `static` method, and the name resolves to a field or method, then that field or method must be `static`

For example:

```
class X {
  private int x;
  int g() { return x; }
}

class Y extends X {
  int f() { return x; }
  static int h() { return g(); }
  int f2() { return Y.h(); }
}
```

In the above code, the name `x` (in the body of `f`) has a semantic error because `x` is not accessible from `Y`. In method `h`, the name `g` causes a compilation error because `g` is not `static` and `h` is. In method `f2`, the name `Y` is resolved to class `Y` and expression `Y.h()` is type-safe.

6.3.12 Checking Constructor Calls

Within the body of a constructor for class `X`, the first statement (and only the first statement) may be a call of the form `super`(+...)`.` This call invokes the the constructor for the superclass of `X` (see section 4.4.1 for details), and is checked like any other constructor call.

6.4 Checking Statements

Statements have no types themselves; instead, the requirement for a statement to be type-safe is that any expressions contained within the statement be type-safe and have the appropriate type.

6.4.1 Checking Empty Statements

Empty statements are inherently type-safe, and no checking needs to be done.

6.4.2 Checking Expression Statements

A statement of the form *e*`;` (where *e* is an expression) is type-safe if and only if *e* is type-safe. The type of *e* is irrelevant. For example, `2;` is a valid statement.

6.4.3 Checking Declaration Statements

A declaration of the form T `v` is type-safe if T is valid. A declaration of the form T `v =` e is type-safe if T is valid, e is type-safe, and the type of e is a subtype of T. The declaration must also satisfy all the restrictions listed in section 6.2.

6.4.4 Checking `if` Statements

An `if` statement of the form `if (` e_1 `)` S_1 `else` S_2 is type-safe if and only if e_1 is a type-safe expression of type `boolean`, and statements S_1 and S_2 are type-safe.

6.4.5 Checking `while` Statements

A `while` statement of the form `while (` e_1 `)` S_1 is type-safe if and only if e_1 is a type-safe expression of type `boolean`, and statement S_1 is type-safe.

6.4.6 Checking `return` Statements

A return statement of the form `return ;` is type-safe if and only if the method containing the `return` has a return type of `void`. A return statement of the form `return` e`;` is type-safe if and only if e is type-safe and has a type of T, and the return type of the method containing the `return` is a subtype of T.

6.4.7 Checking `continue` and `break`

The `continue` and `break` statements can only appear within the body of a `while` loop. There is no type checking to be done on them.

6.4.8 Checking Block Statements

A block statement is type-safe if and only if all of its sub-statements are type-safe.

For methods with a non-`void` return type, the last statement in the top-level block for the method must be a `return` statement.

7 Evaluation

This section specifies the results of evaluating a Decaf program in informal terms. First, the model of execution for Decaf programs will be presented. Second, the effects of evaluating expressions will be discussed. Finally, the effects of executing statements will be described.

7.1 Decaf Execution Model

The Decaf runtime environment has two types of storage:

- *Activation records*, that are allocated every time a method is invoked. The activation record for a method holds the values of all local variables and all the formal parameters.
- The *heap*, which stores objects and arrays.

All Decaf variables and fields hold *values*. A value is either an instance of a non-`void` primitive type (`int`, `boolean`, or `char`), or a *reference* to an object or array stored in the heap. We will use the following notation to represent values:

- The integer value x is represented by x_{int}, e.g., 5_{int} represents 5, 72_{int} represents 72, and so on.
- The character value `'c'` is represented by $\texttt{'c'}_{char}$ or 99_{char} (the ASCII code).
- The boolean value $true$ is represented by 1_{bool} and $false$ by 0_{bool}.
- A reference to the element stored at location i in the heap is represented by i_{ref}. For example, 2_{ref} refers to the item stored at location 2.
- $?_T$ represents an uninitialized value of type T.

We will use the following notation to represent objects stored in the heap. An object that was created as an instance of class `C`, and has an integer field `x` whose value is 5, and a boolean field `y` whose value is `false` will be represented by:

$$Obj_{\texttt{C}}[\texttt{x}_{\texttt{C}} \mapsto 5_{int}, \texttt{y}_{\texttt{C}} \mapsto 0_{bool}]$$

This notation keeps track of the original class of the object plus all the fields. The class in which each field is defined is listed as part of the field name in order to handle hidden fields. For instances of class `String`, we'll use the slightly abbreviated notation:

$$Obj_{\texttt{String}}[\text{``}xxx''] $$

to represent the string literal `"xxx"`.

An array of type T stored in the heap will be represented by:

$$Arr_T[v_0, v_1, \ldots v_{n-1}]$$

For example:

$$Arr_{\texttt{int}}[3_{int}, 11_{int}, -872_{int}]$$

represents an integer array of one dimension and three elements containing the values 3, 11, and -872. Multidimensional arrays are stored as arrays whose elements are references to sub-arrays. For example the array:

2	4	5
7	1	0

would be stored in memory as:

$$Arr_{A(\texttt{int})}[i_{ref}, j_{ref}]$$

where the ith and jth elements in the heap are:

Location	**Value**
i	$Arr_{\texttt{int}}[2_{int}, 4_{int}, 5_{int}]$
j	$Arr_{\texttt{int}}[7_{int}, 1_{int}, 0_{int}]$

Activation records are organized as follows. The storage cells of the activation record are numbered using consecutive integers, starting with 0. The formal parameters to the method will be stored at the beginning of the activation record, followed by all the local variables. For example, consider:

```
static int
f(int x, boolean y)
{
  int z = 22;
  return x + z;
}
```

When `f(87, false)` is invoked, an activation record will be created with the following values:

Offset	**Value**
0	87_{int}
1	0_{bool}
2	$?_{int}$

Where 0 is the offset of `x`, 1 is the offset of `y` and 2 is the offset of `z`. `z` is uninitialized until the execution of the first line of `f`.

7.1.1 Start Up

The evaluation of a Decaf program starts with a call to the `main` subroutine. Exactly one of the classes must contain a method

```
  public static void main(String argv[]) { ... }
```

It is an error if there is more than one `main` or no `main`.

The initial state of the heap and stack is empty. Before `main` starts, a `String` array is created containing the command-line arguments used to invoke the Decaf program from the operating system shell. This array is then passed to `main` as an argument. So, if the program is started with the command:

```
% myprog -c file.out
```

The initial contents of the heap will be (the choice of storage locations is arbitrary):

0	$Obj_{\texttt{String}}$[“myprog”]
1	$Obj_{\texttt{String}}$[“-c”]
2	$Obj_{\texttt{String}}$[“file.out”]
3	$[0_{ref}, 1_{ref}, 2_{ref}]$

The value 3_{ref} will be passed to main. See section 7.2.4 for details on how methods are called. If main has a single local variable `x`, then the activation record for `main` will look like:

Offset	**Value**
0	3_{ref}
1	? (`x` is stored here)

7.2 Evaluating Expressions

The evaluation of an expression does two things: (1) it yields a value, and (2) it possibly modifies the state of the heap and current activation record. The value and effects of each kind of expression is covered in the following sections. We assume there is an ample supply of "scratch" storage locations to hold the intermediate values computed while evaluating expressions. Some implementations could use registers, while others might use a stack to hold temporary values.

7.2.1 Evaluating Literals

The result of evaluating an integer literal is the integer value represented by the literal (using base-10). There is no effect on the heap or activation record. For example, evaluation of `5132` results in the value 5132_{int} with no side-effects.

The result of evaluating a character literal is the ASCII code of the character. For example, evaluation of `'x'` has a resulting value of 120_{int}. There is no effect on the heap or activation record.

The result of evaluating a boolean literal is the boolean value represented by the literal. Hence, evaluating `true` results in 1_{bool}, and evaluating `false` results in 0_{bool}. The heap and activation record are not affected.

The result of evaluating a string literal is a reference to an object of class `String` (stored in the heap) whose value is the string represented by the literal. This string could be pre-allocated or it could be allocated at the time the literal is evaluated. A reference to the location of the string is the result of the expression. For example, the evaluation of `"car"` would result in i_{ref}, where the ith cell of the heap contains $Obj_{\texttt{String}}[\text{“car”}]$.

The result of evaluating the `null` literal is a special reference that doesn't refer to any valid heap location. We will use the notation $null_{ref}$ to represent this location. The results of attempting to dereference $null_{ref}$ are undefined (typically, a runtime error would result). Evaluation of `null` results in $null_{ref}$.

7.2.2 Evaluating Unary Expressions

The expression `!`e evaluates to 1_{bool} if e evaluates to 0_{bool}, and 0_{bool} if e evaluates to 1_{bool}.

The expression `-`e negates the integer expression e. If e evaluates to n_{int}, then `-`e evaluates to $(-1 \cdot n)_{int}$.

The expression `+`e takes the absolute value of the integer expression e. If e evaluates to n_{int}, and n is non-negative, then `-`e evaluates to n_{int}. If e evaluates to n_{int}, and n is negative, then `-`e evaluates to $(-1 \cdot n)_{int}$.

7.2.3 Evaluating Binary Expressions

Evaluation of the mathematical operators is fairly straightforward. If the result of the evaluation is too large to fit in an `int`, then the result becomes undefined. The order of evaluation is left-to-right for all binary expressions.

Numerical Operators `+ - * /` and `%`

The expression e_1 `+` e_2 computes the sum of integer expressions e_1 and e_2. If e_1 evaluates to x_{int} and e_2 evaluates to y_{int}, then the result of e_1 `+` e_2 is $(x + y)_{int}$. Similarly, `-` performs subtraction, `*` does multiplication, `/` performs division (rounded down towards $-\infty$), and `%` computes the remainder. Hence, the result of evaluating `1 + ((2 * (3 - (4 % 2))) / 4)` is 2_{int}.

Boolean Operators `&&` and `||`

The expression e_1 `&&` e_2 evaluates to 1_{bool} if e_1 and e_2 both evaluate to 1_{bool}, and 0_{bool} otherwise. The expression e_1 `||` e_2 evaluates to 0_{bool} if e_1 and e_2 evaluate to 0_{bool}, and 1_{bool} otherwise.

Equality Operators `==` and `!=`

The evaluation of e_1`==`e_2 depends on the type of e_1 and e_2:

- If e_1 evaluates to x_{int}, and e_2 evaluates to y_{int} then the result is 1_{bool} if x equals y and 0_{bool} otherwise.
- If e_1 evaluates to x_{bool}, and e_2 evaluates to y_{bool}, then the result is 1_{bool} if x equals y and 0_{bool} otherwise.
- If e_1 evaluates to y_{ref}, and e_2 evaluates to y_{ref}, then the result is 1_{bool} if x equals y and 0_{bool} otherwise. In other words, the heap storage location must be the same for both references. The value of the item referred to is not checked, so a copy of an object will have result 0_{bool} when compared with the original.
- The evaluation of $e_1 verb+! = +e_2$ results in 1_{bool} if evaluating e_1`==`e_2 would result in 0_{bool} and 0_{bool} otherwise.

Relational Operators `<=` `<` `>` and `>=`

The relational operators only work with integer subexpressions. As always, evaluation is done left-to-right. Assuming e_1 evaluates to x_{int} and e_2 evaluates to y_{int}, the result is:

Operator	**Comparison**
$e1$ `<=` $e2$	1_{bool} if and only if x less than or equal to y
$e1$ `<` $e2$	1_{bool} if and only if x less than y
$e1$ `>` $e2$	1_{bool} if and only if x greater than y
$e1$ `>=` $e2$	1_{bool} if and only if x greater than or equal to y

Assignment Operator `=`

The evaluation of e_1 `=` e_2 depends on the syntactic form of e_1. There are four cases to consider:

(Case 1)	e_1 is the name of a local variable
(Case 2)	e_1 is the name of a field
(Case 3)	e_1 has the form e_1'`.f`
(Case 4)	e_1 has the form e_{11}`[`e_{12}`]`

(Case 1): If e_1 is the name of a local variable, then it must have some storage offset i in the current activation record. The result v of evaluating e_2 is stored at offset i in the current activation record, and v is final result of the expression. For example:

```
  static int f(int x)
  {
    int y = 2;
    int z = 3;
    z = x * y * z; \\ line 3
    return z;
  }
```

If `f` is invoked with an actual argument of `23`, then just before line 3 is executed, the state of the current activation record will be:

Offset	**Value**
0	23_{int}
1	2_{int}
2	3_{int}

After line 3 executes the state of the activation record will be:

Offset	**Value**
0	23_{int}
1	2_{int}
2	138_{int}

(Case 2): If e_1 is the name of a field, then the expression is evaluated as if it was `this.`$e_1 = e_2$, using (Case 3) rules.

(Case 3): If e_1 is of the form e'_1`.f`, then the expression e'_1 is evaluated to yield a reference i_{ref} to an object. The result v of evaluating e_2 is then stored in the field `f` of the object referred to by i_{ref}. For example, consider the class:

```
class X {
  int myX;
  X(int xi) { myX = xi; }
}
```

Suppose that the following code sequence executes:

```
   X x = new X(5);    // line 1
   X.myX = 10;        // line 2
```

After line 1 executes, the value of x will be i_{ref}, where the ith cell of the heap contains:

$$Obj_{\mathtt{X}}[\mathtt{myX}_{\mathtt{X}} \mapsto 5]$$

After evaluation of the expression `x.myX = 10`, cell 5 of the heap will be modified so that it contains:

$$Obj_{\mathtt{X}}[\mathtt{myX}_{\mathtt{X}} \mapsto 10]$$

(Case 4): If e_1 is of the form e_{11}`[`e_{12}`]`, then the expression e_{11} is evaluated to yield a reference a_{ref} to an array, and the expression e_{12} is evaluated to yield an integer index i_{int}. The value v resulting from the evaluation of e_2 is then stored in the ith cell of the array (using zero as the index of the first element). If i is not a valid index (less than zero or greater than than the size of the array minus one), then a runtime error occurs. The final result of the expression is v. For example:

```
int A[] = new int[5];                    // line 1
A[0] = A[1] = A[2] = A[3] = A[4] = 0;    // line 2
A[2] = 3;                                // line 2
```

After line 1 is executed, the value of `A` will be i_{ref}, where the ith cell of the heap contains:

$$Arr_{\mathtt{int}}[?_{int}, ?_{int}, ?_{int}, ?_{int}, ?_{int}]$$

Line 2 will assign 0 to all cells of the array referred-to by the value of `A`, modifying the ith cell of the heap to:

$$Arr_{\mathtt{int}}[0_{int}, 0_{int}, 0_{int}, 0_{int}, 0_{int}]$$

Line 3 will modify the third cell of the array, so that the ith heap cell will contain:

$$Arr_{\mathtt{int}}[0_{int}, 0_{int}, 3_{int}, 0_{int}, 0_{int}]$$

7.2.4 Evaluating Method Invocations

There are two types of method. `static` methods are evaluated like functions in a traditional non-object-oriented language like Pascal or C. Non-`static` methods are bound dynamically (using the runtime lookup discussed in section 2), and have a special "hidden" formal argument named `this`. For non-`static` methods, `this` is always the first formal parameter. The second and subsequent parameters (if any) contain the formal parameters declared in the program.

The rules for method invocation are broken into five cases:

(Case 1) `f(`$e_1, e_2, \ldots e_n$`)` and `f` is `static`
(Case 2) e`.f(`$e_1, e_2, \ldots e_n$`)` and e`.f` is `static`
(Case 3) `super.f(`e_1`, `e_2`, `$\ldots e_n$`)` and `super.f` is `static`
(Case 4) `f(`e_1`, `e_2`, `$\ldots e_n$`)` and `f` is non-`static`
(Case 5) e`.f(`$e_1, e_2, \ldots e_n$`)` and e`.f` is non-`static`
(Case 6) `super.f(`$e_1, e_2, \ldots e_n$`)` and `super.f` is non-`static`

(Case 1) In this case, the method to be called is fixed at compile time. The arguments are evaluated from left to right, yielding values $v_1 \ldots v_n$. A new activation record is created with locations $0 \ldots n-1$ holding the argument values (location i holds value v_{i+1}. The body of the method is evaluated in the context of the new activation record. The activation record is thrown away when evaluation of the method body returns. The result of the evaluation is the return value of the method.

(Case 2) e is evaluated, and the result of the evaluation is thrown away. After that, the evaluation proceeds using the rules for (Case 1).

(Case 3) Evaluated exactly like (Case 1).

(Case 4) Evaluated like the expression `this.f(`$e_1, e_2, \ldots e_n$`)`, using the rules for (Case 5).

(Case 5) e is evaluated, resulting in a reference i_{ref} to an object. The actual arguments $e_1 \ldots e_n$ are evaluated left-to-right, resulting in values $v_1 \ldots v_n$. The dynamic class of i_{ref} is used to dynamically select the method m to be called. An activation record for m is created. The first formal parameter (location 0) holds i_{ref}. For $i = 1 \ldots n$, activation record location i holds v_i. The body of the method m is evaluated in the context of the new activation record. The activation record is thrown away when evaluation of the method body returns. The result of the evaluation is the value returned by m.

(Case 6) Even though the method is not `static`, the m method to be called is bound at compile time to the `f` in the superclass of the class containing the `super` expression. An activation record for m is created. The first formal parameter (location 0) holds i_{ref}. For $i = 1 \ldots n$, activation record location i holds v_i. The body of the method m is evaluated in the context of the new activation record. The activation record is thrown away when evaluation of the method body returns. The result of the evaluation is the value returned by m.

Consider the following class:

```
class X {
  private int x;
  private int y;

  static int f() { return x + y; }
  int x() { return x; }
  int y() { return y; }
  void setX(int xn) { x = xn; }
  int setY(int yn) { y = yn; }
  X(int xi, int yi) { x = xi; y = yi; }
}
```

Assume that `x` is a variable whose type is $C(\texttt{X})$ and whose value is 187_{ref}. The expression `x.setX(11)` would result in the creation of the activation record:

Offset	**Value**
0	187_{ref}
1	11_{int}

The expression `x.f()` would result in the creation of an empty activation record.

7.2.5 Evaluating Variable Names

The result of evaluating the name of a local variable is the value currently stored in the variable. The activation record offset of the variable is known at compile time, and the value stored in that location is result of the expression. For example:

```
static int f(int x)
{
  int y = x + 5;
  return y;
}
```

If `f` is called with an argument of 5, then just before the `return` is executed the activation record will look like the following:

Offset	**Value**
0	5_{int}
1	10_{int}

The name `y` in `return y` refers to the local variable `y`, whose offset in the current activation record is 1. When `y` is evaluated, the value stored at offset 1 is returned, resulting in a value of 10_{int}.

7.2.6 Evaluating Field Names

The result of evaluating the name of a field f is the same as evaluating the expression `this.`f.

7.2.7 Evaluating Method Names

No computation is done for `static` methods, since the method to call is known at compile time. For dynamic methods, we'll be vague for now and assume there is a machine instruction to lookup methods at runtime using the dynamic class of an object.

7.2.8 Evaluating Dot Expressions

Expressions of the form e`.f` are evaluated by first evaluating e, which must result in an reference to an object. The value stored in the field `f` of the object is then returned (if there is more than one `f`, then the rules of section 5 are used to determine which `f` is being referred-to). The offset of each field is determined at compile time. No dynamic binding of `f` occurs. For example:

```
class Y {
  int y;

  int getYsY() { return y; }
  Y(int yi) { y = yi; }
}

class X extends Y {
  int x;
  int y;

  int getXsY() { return y; }
  X(int yyi, int, xi, int yi} { Y(yyi); x = xi; y = yi; }
}
```

After the statement `X x = X(1, 2, 3);` is executed, the value of `x` will be i_{ref}, where the ith cell of the heap contains

$$Obj_{\texttt{X}}[\texttt{y}_{\texttt{Y}} \mapsto 1_{int}, \texttt{x}_{\texttt{X}} \mapsto 2_{int}, \texttt{y}_{\texttt{X}} \mapsto 3_{int}]$$

If `x.getYsY()` is evaluated, the activation record for `getYsY` will look like:

Offset	Value
0	i_{ref}

When the expression `y` in `getYsY` is evaluated, the value of $\texttt{y}_{\texttt{Y}}$ in the $Obj_{\texttt{Y}}$ stored at location i_{ref} in the heap is returned, yielding 1_{int} as the return value of `getYsY`. On the other hand, if `getXsY` is called, then return value would be the value of $\texttt{y}_{\texttt{X}}$, which results in a return value of 3_{int}.

7.2.9 Arrays and `.length`

There is one special form of dot expression: an expression of the form e`.length`, where e has an array type, evaluates to the number of elements in the first dimension of the array. For example, `(new int[5]).length` evaluates to 5.

7.2.10 Evaluating `this`

`this` is a "hidden" formal parameter of all non-`static` methods. The result of evaluating `this` is the value of the first formal parameter (stack location

0). `this` is not defined for static methods.

7.2.11 Evaluating `super`

`super` is evaluated as if it were `this`.

7.3 Executing Statements

Statements have no resulting value. Hence, we will use the term *execution* instead of evaluation when discussing the semantics of statements. When a statement is executed, it usually modifies the state of the heap or activation record in some manner. The rules for execution of each kind of statement are presented in the next sections.

7.3.1 Executing Empty Statements

The empty statement (`;` by itself) has no effect. Hence nothing needs to be done to execute it.

7.3.2 Executing Declarations

Execution of a declaration with no initialization has no effect.

Execution of a declaration with initialization is treated just like assignment to a local variable. In other words, the declaration `int x = 2;` is executed by evaluating the expression `x = 2`, and ignoring the result of the evaluation. Note that these rules imply the initialization should re-occur every time the flow of control passes over the declaration.

7.3.3 Executing `if` Statements

A statement of the form `if (`$everb+$`)` $+ S_1$ `else` S_2 is evaluated by first evaluating e. If e evaluates to 1_{bool}, then S_1 is executed. If e evaluates to 0_{bool}, then S_2 is executed. A statement of the form `if (`e`)` S_1 is executed as if it were `if (`e`)` S_1 `else ;`.

7.3.4 Executing `while` Statements

A statement of the form `while (`e`)` S_1 is executed in the following steps:

1. Evaluate e.
2. If the result of evaluating e is 1_{bool}, then execute S_1. The execution then starts over again at step 1.
3. If the result of evaluating e is 0_{bool}, then execution of the `while` statement is finished.

7.3.5 Executing `break` Statements

A `break` statement can only appear within a `while` loop. Execution of the break statement causes the innermost `while` loop to immediately terminate. Flow of control jumps to the statement immediately after the end of the `while` loop.

7.3.6 Executing `continue` Statements

A `break` statement can only appear within a `while` loop. Execution of the break statement causes the innermost `while` loop to immediately start over. Flow of control jumps to top of the `while` loop (the condition is re-evaluated).

7.3.7 Executing `return` Statements

A `return` statement causes the current method to immediately terminate. If the `return` statement has an associated expression e, then e is evaluated and the result is used as the value of the expression that invoked the current method.

7.3.8 Executing Block Statements

A block statement of the form `{` S_1 `;` S_2 `;` $\ldots S_n$ `;` `}` is executed by sequentially executing $S_1 \ldots S_n$. First S_1 is executed, then S_2, and so on. The flow of execution may be interrupted by a `break`, `continue`, or `return` statement. A block containing no sub-statements (i.e., $n = 0$) is semantically equivalent to the empty statement.