ArchJava Language Reference Manual

This document describes the design of the ArchJava language. Each section introduces a construct, gives an example, motivates the construct, and gives the grammar and necessary semantic rules. To allow programmers to describe static software architecture, ArchJava adds new language constructs to support components, connections, and ports.

Limitations of the 1.0 release

- This is a robust research prototype: we will support this compiler as best we can, but it should not be considered a production-quality compiler.
- Most javac compiler flags are supported, and will be passed through to the back end javac compiler.
- This release does not yet include source code; contact the authors if you are interested in the source.
- Some features are not yet fully implemented:
  - incomplete specifications
  - Legacy or Evolution mode (the compiler acts as if it were always in legacy and evolution modes)
  - Certain static and dynamic checks

1. Components and Ports

1.1. Components

Example: a parser component

```java
public component class Parser {
    public port in {
        provides void setInfo(Token symbol,
                               SymTabEntry e) { ... }
        requires Token nextToken()
                   throws ScanException;
    }
    public port out {
        provides SymTabEntry getInfo(Token t);
        requires void compile(AST ast);
    }

    public void parse(String file) {
        Token tok = in.nextToken();
        AST ast = parseFile(tok);
        out.compile(ast);
    }

    private void parseFile(Token lookahead) { ... }
    public SymTabEntry getInfo(Token t) { ... }
    ...
}
```

A component is a special kind of object that communicates with other components in a structured way. Components are instances of component classes. Component classes can contain port declarations and connect declarations in addition to the declarations that ordinary classes can contain.

Components can be dynamically instantiated using the same new syntax used to create ordinary objects. At creation time, each component records the component instance that created it as its parent component. For components that are instantiated outside the scope of any component instance, the parent component is null.

A component will eventually be garbage collected if there are no references to the component or its connections reachable from garbage collection roots.
Semantic Rules

1.1.1. Components may not have non-static inner classes, except in the compiler’s legacy mode
This is because inner classes have a reference to their enclosing component, and so inner class objects can escape and allow any part of the program to access a component, violating communication integrity. The legacy mode exception is for interoperability with legacy Java libraries like Swing, which make heavy use of inner classes.

1.1.2. Components cannot extend ordinary classes (other than Object) or implement interfaces that declare methods, except in the compiler’s legacy mode
Without this rule, components could escape their parent component as a superclass/superinterface object, and inherited methods could be invoked from any part of the program, violating communication integrity. The legacy mode exception is for interoperability with legacy Java libraries like Swing, which require overriding ordinary classes. Interfaces that do not declare methods can still be used to define final constants common to several components.

1.1.3. Ordinary classes may not declare fields of component or component array type.
Since ordinary objects can be shared arbitrarily within the program, they cannot have static references to components; otherwise, any part of the program could call an object method, which could invoke a component method, violating communication integrity.

1.1.4. Component and component array types may not appear in the signatures of component methods unless they are private or protected
This rule prevents components from being passed outside the scope of their parent component, violating communication integrity. The rule allows components to be passed around within a single component.

1.1.5. Fields of component or component array type within component classes must be private or protected
This rule prevents components from escaping the scope of their parent component, violating communication integrity. The rule allows components to be stored in fields and arrays within a single component.

1.1.6. Private and protected component members can only be accessed on this or super, not on other components of the same class
Because private and protected component methods may have component types in their signatures, calling them on another component instance could allow a component to escape its parent component, violating communication integrity.

1.1.7. Casts to component type are prohibited in ordinary classes
This rule prevents a component from being passed as an object to another part of the system, then downcast to a component type so its methods can be invoked, violating communication integrity.

1.1.8. Casts to component type inside a component throw a ComponentCastException if the cast expression’s parent is not equal to this.
It is safe to downcast to a component type within the scope of its parent component. A run-time check verifies that the required condition is true.

1.1.9. Casts to component array types are prohibited everywhere
The above parent component check cannot be done efficiently for an array of components without changing the Java virtual machine; therefore, all casts to component array type are prohibited to ensure communication integrity. A workaround is to copy components from an object array to a new component array, casting each element individually to component type.

1.1.10. Incomplete components may leave bodies of methods unspecified; these methods will be compiled with a body that simply throws an UnimplementedMethodException.
This rule is intended to support partial system implementation early in the software lifecycle.

1.1.11. Component classes otherwise have all the same semantics as ordinary classes
type_decl ::= ...
    |   component_decl
component_decl ::= modifier* component_class id component_body
component_body ::= { component_body_decl* }
component_body_decl ::= class_body_decl
    |   port_decl
    |   subcomponent_decl
    |   connect_decl

1.2. Ports

A component instance communicates with external components through ports. A port represents a logical communication channel between a component instance and one or more components that it is connected to. If a component participates in more than one logical communication channel, it specifies a port for each different channel.

Ports declare three sets of methods, specified using the requires, provides, and broadcasts keywords. Provided methods can be invoked by other components connected to the port. Provided methods may be implemented with a separate method definition outside the port (setInfo in the parser example above), or given a body inside the port (getInfo). The component can invoke a disjoint set of required methods through the port. Each required method is implemented by a component that the port is connected to. Broadcast methods are just like required methods, except that they may be connected to an unbounded number of implementations.

Each port is a first-class object that implements its required and broadcast methods, so a component can invoke these methods directly on its ports. For example, the parse method calls nextToken on the parser’s in port. These calls will be bound to external components that implement the appropriate functionality.

Semantic Rules

1.2.1. Ports may be defined only in component classes, except that in the compiler’s evolution mode, they may be defined in ordinary classes as well.

Allowing components to communicate with objects through ports could allow violations of communication integrity, since the connected objects could be passed anywhere in the program and invoked arbitrarily. However, such ports are useful when evolving a legacy Java system incrementally to express its architecture in ArchJava.

1.2.2. Required and broadcast methods may not be given an implementation

Required and broadcast methods support invoking the methods of connected objects, not the methods of the current object.

1.2.3. Provided methods in a port may not be invoked directly. For example, calls such as aComponent.aPort.aMethod() is illegal if aMethod is provided.

If a provided method is of a generic nature, it should be defined in the component. Calling provided methods defined inside a port directly is problematic, because there would be no definition for the sender variable (see below).

1.2.4. Required and broadcast methods in a port may only be invoked directly from within the component defining the port, i.e. on an expression of the form this.portname.methodname(args)

Required methods are intended for a component to communicate with external components.

1.2.5. Provided methods defined in a port may not be invoked on the component. For example, if aMethod is defined inside aPort, then a call such as aComponent.aMethod() would be illegal.

This rule avoids the potentially ambiguous situation when two methods with the same name and signature are provided on different ports. Which one would be intended? Also, there would be no definition for the sender variable (see below).
1.2.6. Each method name and signature can appear at most once inside a port. For example, a port may not both require and provide the same method.

The semantics of connections would be very ambiguous and confusing if this were allowed, although it may be useful for broadcast messages when the sender wants to be notified of the broadcast. A workaround for the broadcast case is defining the broadcast and provided methods in different ports, which would be connected together by the parent component.

1.2.7. Component and component array types may not appear in the signatures of port methods

*Same motivation as prohibiting these types in public or private methods.*

1.2.8. Calling a required or broadcast method before the enclosing port has been connected will result in throwing a

*UnconnectedException*

This provides a well-defined semantics for situations when a dynamically created component has not yet been connected to its neighbors.

1.2.9. Broadcast methods must have a return value of **void**

This sidesteps the question of what to return if the methods invoked in a broadcast return different results. A workaround is to implement a custom connector that implements broadcast manually, then combines the return values in a system-specific way.

```plaintext
port_decl ::= port_modifier* port interfaceopt id port_body

port_body ::= { port_body_decl* }
  | ; ;

port_body_decl ::= port_method_modifier* method_declaration
  | port_method_modifier* method_header ;

port_method_modifier ::= method_modifier
  | provides
  | requires
  | broadcasts ;

port_modifier ::= public
  | protected
  | private ;
```
1.3. Port Interfaces

```java
public component class WebServer {
    private final Router r = new Router();
    connect r.request, create;
    connect pattern Router.workers, Worker.serve;

    public void run() { r.listen(); }

    private port create {
        provides r.workers requestWorker() {
            final Worker newWorker = new Worker();
            r.workers connection = connect(r.workers, newWorker.serve);
            return connection;
        }
    }
}

public component class Router {
    public port interface workers {
        requires void httpRequest(InputStream in, OutputStream out);
    }
    public port request {
        requires this.workers requestWorker();
    }
    public void listen() {
        ServerSocket server = new ServerSocket(80);
        while (true) {
            Socket sock = server.accept();
            this.workers conn = main.requestWorker();
            conn.httpRequest(sock.getInputStream(), sock.getOutputStream());
        }
    }
}
```

Often a single component participates in several connections using the same conceptual protocol. For example, the `Router` component in the web server communicates with several `Worker` components, each through a different connection. A `port interface` describes a port that can be instantiated several times to communicate through different connections at run time.

Each port interface defines a type that includes all of the required and broadcast methods in that port. A `port interface type` combines a port’s required interface with an `instance expression` that indicates which component instance the type allows access to. For example, in the `Router` component above, the type `this.workers` refers to an instance of the `workers` port of the current `Router` component (in this case, `this` would be inferred automatically if it were omitted). The type `r.workers` refers to an instance of the `workers` port of the `r` subcomponent. This type can be used in method signatures such as `requestWorker` and local variable declarations such as `conn` in the `listen` method. Required methods can be invoked on expressions of port interface type, as shown by the call to `httpRequest` within `Router.listen`.

When a connection object is read from a collection, it can be cast to the appropriate port interface type so that methods can be invoked through the connection. To enforce communication integrity, ArchJava verifies that the component instance in the port interface type is one of the components connected by the connection object (throwing a `ConnectionCastException` if it is not).

**Semantic Rules**

1.3.1. Required and broadcast methods may not be invoked directly on a port interface

Since port interfaces represent a protocol that may be used in multiple connections, programmers must specify the intended connection by invoking the method on a connection object instead.

1.3.2. Provided methods may only be invoked on expressions of port interface type where the instance expression is `this`.

This ensures that only a connected component may invoke methods on a connection.
1.3.3. Every ordinary port is an object of its own port interface type, with the instance expression `this`. This allows ordinary ports and port interfaces to be treated in a uniform way.

1.3.4. Port interface types may have instance expressions of the form `this`, a `final` variable currently in scope, or a `final` field.
   These expression forms were chosen since they are immutable and easy to reason about. Mutable expressions would allow violations of communication integrity because when the instance expression changes, the invariant that the component identified by the instance expression participates in the connection would be violated. Note that `final` method arguments can be used as instance expressions in method signatures.

1.3.5. Port interface types are a subtype of Object, so an expression of port interface type can be passed where an expression of type Object is required. This allows port interfaces to be stored in generic data structures.

1.3.6. Casts to port interface type throw a `ConnectionCastException` if the instance expression is not a component with the relevant port connected as part of the connection object in the cast expression. This prevents a connection object from being used to communicate with components in ways that would violate communication integrity. A runtime check verifies that the required condition is true.

1.3.7. Casts to port interface array types are prohibited everywhere. The above check cannot be done efficiently for an array of components without changing the Java virtual machine; therefore, all casts to port interface array type are prohibited to ensure communication integrity. A workaround is to copy port interfaces from an object array to a new port interface array, casting each element individually to the required port interface type.

Rules for matching port interface types

1.3.8. When calling a method on a `final`, component-typed field `s` with an argument of port interface type `s.p`, the argument will match the port interface type `this.p` in the method’s declaration. This makes it easy to pass connection objects to subcomponents in a type-safe way via direct method calls.

1.3.9. When connecting a private port `p` to a port `sp` of a subcomponent in a `final` field `s`, the instance expression `s` in a port interface type of a method in port `p` will match the instance expression `this` in a port interface type of a method with the same name and an otherwise matching signature in port `sp`. This makes it easy to pass connection objects to subcomponents in a type-safe way via calls and returns through ports (as demonstrated in the web server example).

1.3.10. When passing both a component and a port interface type that has that component as its instance expression to a method, the port interface type of the actual argument will match a port interface type of the formal where the formal’s instance expression is the `final` formal argument that the component was passed to. Although component references cannot be passed from one component to another, this mechanism can be used to pass port interface types to objects, or between methods within a single component.

1.3.11. Otherwise, an exact match is required
   Just as with Java’s inheritance rules—an overriding method must have an identical signature to the method it overrides.

1.4. Sender

Provided methods can obtain the connection object through which the method call was invoked using the `sender` keyword. Defined in every provided method that is implemented inside a port, the `sender` variable holds a connection object that implements the port interface type of the enclosing port, with `this` as the instance expression.

Semantic Rules

1.4.1. The `sender` keyword can only appear in the scope of a port or a port interface
   Methods defined outside the scope of a port may be called directly (not through a connection) and so there might be no `sender` object.
2. Composition

2.1. Subcomponents

```java
public component class Compiler {
  private final Scanner scanner = new Scanner();
  private final Parser parser = new Parser();
  private final CodeGen codegen = new CodeGen();

  connect scanner.out, parser.in;
  connect parser.out, codegen.in;

  public static void main(String args[]) {
    new Compiler().compile(args);
  }

  public void compile(String args[]) {
    // for each file in args do:
    ...parser.parse(file);...
  }
}
```

A subcomponent is a component instance that is instantiated inside another component class. Components can invoke methods directly on their subcomponents. However, subcomponents cannot communicate with components external to their containing component. Thus, communication patterns among components are hierarchical. For example, the compiler component class above creates scanner, parser, and code generator subcomponents, and stores them in private final fields of the compiler.

2.2. Connections

The `connect` primitive connects two or more subcomponent ports together, binding each required method to a provided method with the same name and signature.

Semantic Rules

2.2.1. Connections may appear only in component classes

*Connections are intended to express software architecture, which is embedded in component classes in the ArchJava language.*

2.2.2. Connections are instantiated immediately after the call to `super()` in the enclosing component’s constructor

*This ensures that connections are already instantiated when the enclosing component’s constructor runs.*

2.2.3. Connections may connect port interfaces as well as ports,

*Although port interfaces cannot invoke required methods directly, they can provide methods, and can invoke required methods on the sender connection object that is passed to provided methods.*

2.2.4. Only `final` component fields may participate in a connection, except that final object fields may be connected in evolution mode

*It is difficult to reason about connecting a non-final field to other objects, and connecting objects into an architecture can violate communication integrity, since aliases to objects may exist throughout the program. However, connections to objects are useful temporarily when evolving a legacy Java system incrementally to express its architecture in ArchJava.*

2.2.5. Private and protected ports may not participate in outside connections

*This would violate the semantics of private by exposing private/protected ports to external components.*

2.2.6. Public and package ports may not participate in connections internal to the port’s defining component (although they may be glued to implementations as described below)

*Since public and package ports may participate in external connections, if they also participated in ordinary internal connections, there might be ambiguity—for example, if one of the port’s required methods is provided both by an internal and an external connection, which one should be called? This rule avoid this ambiguity.*
2.2.7. For each required method in connected ports, there must be exactly one corresponding provided method in some other connected port

Each required method must have an implementation that will be called when the required method is invoked; however, if there were more than one implementation, it would be unclear which one should be called.

2.2.8. Any number of required methods may be connected to a single provided method

This supports the common case where one component provides a service that is used by several other components.

2.2.9. A broadcast method may be connected to zero or more provided methods

Broadcast methods are for notification purposes, and have no return value. All the components that are listening to the notification will be notified; the number of connected components does not matter.

2.2.10. Broadcast semantics: connected provided methods will be called sequentially in an undefined order. Return values are not an issue because broadcast methods must return void. The invocation of the broadcast method will return after the last provided method has been called and has returned.

This is the simplest possible choice for broadcast semantics, and closely matches ordinary Java semantics. Programmers who need other semantics, such as a well-defined invocation order, or a return value, or asynchronous, multi-threaded broadcast, can write custom components that serve as “smart connectors.” It would unnecessarily complicate the language to support the many different varieties of broadcast, but the one supported here is sufficient for a wide range of programs.

connect_decl ::= connect patternopt port_list ;
port_list ::= name
| port_list , name

2.3. Connect Expressions

Dynamically created components can be connected together at run time using a connect expression. For instance, the requestWorker method of the web server dynamically creates a Worker component and then connects its serve port to the workers port on the Router.

Port interfaces are instantiated by connect expressions. A connect expression returns a connection object that represents the connection. In the web server example above, the connection object connection implements the types newWorker.serve and r.workers, and can therefore be assigned to a variable of type r.workers.

Semantic Rules

2.3.1. Each connection object implements the port interface types of all the connected ports

Technically, connection objects have a union type that cannot be directly expressed in ArchJava.

2.3.2. Connect expressions may appear only in component classes

This allows them to be matched to a connection pattern in the component class, as described below.

2.3.3. An ordinary port may not be connected twice through connect expressions; if the program attempts to do so, a PortReconnectionException will be thrown and the second connection will not be made.

2.3.4. Each connect expression must match a connection pattern in the enclosing component, as described below

This supports a definition of communication integrity for connect expressions: no dynamically-created components may be connected, except in a way that is matches a connection pattern in their common parent component.

method_invocation ::= ...
| connect_expression
connect_expression ::= connect { argument_list }

connect_decl ::= connect patternopt port_list ;
port_list ::= name
| port_list , name


2.4. Connection Patterns

Communication integrity requires each component to explicitly document the kinds of architectural interactions that are permitted between its subcomponents. A connection pattern is used to describe a set of connections that can be instantiated at run time using connect expressions. For example, connect pattern Router.workers, Worker.serve describes a set of connections between Router and Worker subcomponents.

Each connect expression must match a connection pattern declared in the enclosing component. A connect expression matches a connection pattern if the connected ports are identical and each connected component instance is an instance of the type specified in the pattern. The connect expression in the web server example matches the corresponding connection pattern because the newWorker component in the connect expression is of static type Worker, the same type declared in the pattern (and similarly for r and Router). The port names are identical in the pattern and connect expression.

Semantic Rules (most duplicated from the Connections section)

2.4.1. Connection patterns must follow all of the rules cited above for connections, with the exceptions and additions listed below

   * Connection patterns are the dynamically created analogue of static connections.

2.4.2. Connection patterns are not automatically instantiated; instead, they are instantiated by connect expressions (as described above)

   * Connection patterns specify the kinds of connections that may be made dynamically at run time.

2.4.3. Connection patterns specify ports of a component type as elements to be connected

   * This allows connection of dynamically created components, in addition to those in final fields.

2.4.4. Connection patterns that specify ordinary classes are only permitted in evolution mode

   * Connections to ordinary classes may violate communication integrity because ArchJava does not limit sharing of ordinary classes. However, these connections are useful while evolving a legacy Java system to express its architecture with ArchJava.

2.5. Glue connections

```java
component class Component {
  port main { ... }
  port in {
    requires int getData();
  }
  port out {
    provides int getData();
  }

  private final Subcomponent s = new Subcomponent();

  glue in, out;
  glue main to s.main;
}
```

Glue connections can be used to create pass-through connections to subcomponents or to other ports. In the example above, the component provides the getData method in the out port by forwarding it to the required method in its in port. It also provides methods in its main port by forwarding them to the main port of its subcomponent in field s.

As shown in the example, the glue construct differs from the connect construct because it glues the “inside” of a port to other ports, instead of connecting the “outside” of the port. The semantics of the glue...to... construct can be thought of as one big connection, where the ports in the to section are treated normally, but in the glue section, required methods are treated as provided, implemented provided methods are ignored, and unimplemented provided methods are treated as required. Implemented provided methods in the glue section are simply ignored.
Semantic Rules

2.5.1. Glue connections follow the same rules as ordinary connections, with the exceptions noted below

2.5.2. Required methods of ports in the glue section are considered provided. When connecting a method to a required method of a port in the glue section, the glue code calls the required method of that port.

2.5.3. Provided methods of ports in the glue section that are not implemented either in the port or in the component are considered required. When connecting an unimplemented provided method of a port in the glue section, the method is implemented by calling the method it is connected to.

2.5.4. Provided methods of ports in the glue section that are implemented are ignored.

2.5.5. Only ports of the current component may be specified in the glue section

connect_decl ::= glue port_list to_part opt ;
  to_part ::= to port_list

3. Inheritance

Like other object-oriented languages, ArchJava supports software evolution through inheritance.

3.1. Component Inheritance

A component class can extend another component class. Inheritance of component classes is similar to inheritance of regular classes. Fields, methods, ports, and connections are inherited from the component superclass. Component subclasses may add new fields, methods, ports, and connections between newly added component fields. As in Java, provided methods can be overridden, but component subclasses can also override ports.

Ports can be re-declared in component subclasses, with new implementations for some provided methods.

Semantic Rules

3.1.1. New required methods may not be added to inherited ports or to new ports

Adding required methods to an existing port would make the component class non-substitutable for the component superclass, because connections made to the superclass might not provide the subclass’s required methods. Required methods in a new port are also problematic, because the new port might not be connected at all.

3.1.2. New provided methods may not be added to non-abstract existing ports

Adding new provided methods to an existing port might cause ambiguities if these provided methods were required by a connected component, and provided by a different component. There would then be two components providing the same required method, breaking ArchJava’s connection rules.
4. Design Support

```java
incomplete component class Parser {
  public port in {
    requires Token nextToken() throws ScanException;
  }
  public port out { }
}
```

In addition to supporting fully developed architectures, ArchJava supports the iterative development of architectures during the design phase. ArchJava provides support for constructing, visualizing, and typechecking incomplete architectures.

An incomplete component represents an architectural design or a partial implementation. Like abstract classes in Java, incomplete components need not have all of their methods implemented. However, while the abstract keyword indicates a class that will be extended by concrete subclasses, the incomplete keyword indicates a design-time component that will be implemented at a later point in time. Furthermore, incomplete components can be instantiated and linked into an architecture, supporting design-time checking and early architectural prototyping.

The ArchJava compiler can compile and perform all of the usual type checking on incomplete components; the only difference is that methods in an incomplete component need not be implemented. This supports consistency checking of architectural designs. In addition, ArchJava supports incremental implementation, because method bodies can be given to some of the methods in an incomplete component class. If a part of a system is implemented but other parts are left incomplete, the system can be compiled and run. The compiler generates method bodies for unimplemented methods; these bodies throw an UnimplementedMethodException if they are called at run time.

The incomplete keyword may modify any regular or component class. Methods may not be modified with incomplete, nor may ports. Instead, methods inside an incomplete class may use a semicolon in place of a body (as with abstract methods). Ports must always declare all of the methods that are referenced elsewhere in the program, so that architectural typechecking can still be done on incomplete architectures. A class may be both incomplete and abstract, in which case it cannot be instantiated (like abstract classes) but may have unimplemented, non-abstract methods (like incomplete classes).

The example above shows an early design of the Parser component. The Parser can be connected to the other components in a compiler architecture, as long as the connections typecheck properly. The designer has left out the bodies of all methods at this early design stage. If other parts of the system are implemented, they can be executed; however, an UnimplementedMethodException will be thrown if any of the unimplemented methods of Parser are called.