International Journal of Computer and Systems Engineering

Volume 1 | Issue 1 Article 4

July 2020

Declarative Immutability in Java

John Crowley
Boston College, University of Pennsylvania, j.crowley@computer.org

International Journal of Computer and Systems Engineering

Online at https://digitalcommons.fairfield.edu/ijcase/ Published by Fairfield University, **DigitalCommons@Fairfield** © **2020**

Vol. 1, No. 1, 2020

Declarative Immutability in Java

by John D. Crowley

ABSTRACT A pervasive mechanism is proposed to declare that variables and object *instances* are *mutable* or *immutable*, and whether methods mutate instances. These declarations are integral to the type structure of the language and allow the developer to declare, and the reader (and compiler) to determine, if a variable or instance is immutable (even if other instances of the Class are mutable) or that an instance is immutable throughout the application (a *Pure* instance). The concept of the *owner* or *outsiders* of variables and instances is combined with a series of *tags* to declare mutability, and can be enforced during compilation. This provides a more informative definition of the interface for a Class, requires fewer lines of code for implementation, and reduces the runtime overhead of defensive coding (creating clones). In a multi-threaded application, flagging a *Pure* instance to the JVM can provide significant performance improvements by eliminating unnecessary synchronizations. Many of the benefits recognized for Functional Programming are introduced to Java as an optional enhancement.

INDEX TERMS functional programming, immutable, Java, read-only, software development

I. INTRODUCTION

The goal of this research is to make the declaration of mutability an integral part of the type structure of the language. This allows the developer to declare, and the reader (and compiler) to determine, if a variable or object instance is mutable or immutable within a *specific context*, or is immutable throughout the application. It also allows the declaration of *methods* which are read-only and do not mutate the defining instance (or other objects).

The advantages of immutability long recognized by Functional Programming environments [1] are introduced to Java. This supports faster development of new Classes, makes Classes more robust, provides valuable information to the user or maintainer of the Class, and is more efficient at runtime (especially in high volume, multi-threaded applications).

This discussion proposes extensions to the syntax and semantics of Java to:

- Define *outsiders* and *owners* of variables and instance references,
- Define *tags* to specify whether *variables* or the *target instances* of *references* are mutable or immutable, and by *outsiders* or the *owner.*
- Define tags to declare that *methods* and their *parameters* are *immutable*.
- Extend the tags to *types*, *generics,* and the *return values* of methods.

The syntax of *tags* minimizes the textual expansion of the language, are simple to specify by the developer, and allow the reader to quickly determine if a particular variable or instance is mutable or immutable. The defaults are chosen so that all are ***optional*** – an existing Java program will compile and run with no changes, and match existing semantics.

This mechanism can be checked and enforced at compile time, with the exception of runtime support for self-referential immutable objects. Most importantly, this allows instances which are normally mutable to be used immutably within a specific context – without requiring the implementation of separate Classes (for example the separate immutable and mutable *Collections* in Scala), or wrappers such as java.util.*Collections.unmodifiable*…

There have been several previous attempts to control mutability within a language. The *const* keyword has been supported in C++ for many years [2], and ConstJava implemented these semantics in Java [3]. [7] described a major implementation in C# which influenced the *Pure at Runtime* treatment here. Of course, most Functional Programming languages use immutability as a core concept.

Although this discussion utilizes Java as a base, it may be possible to apply the same concepts to other object-oriented languages such as C#, Scala, Swift, or C++, within the syntactic and semantic context of those languages. (C# [12], Scala [6], and Swift might be better targets since the *property* concept is already supported. See *Synchronization Tags*.)

A modified Java compiler has been implemented as a proof of concept for some key features.

II. MOTIVATION AND EXAMPLE

A. Variables

Mechanisms already exist to protect *variables* – they can be declared *private* and accessed through getter/setter methods. A variable may also be declared *final* and prevent any modification. However, there are still unknowns for the reader or user of a Class (unless *final*):

- Is there a *setter* method at all? The current state-of-the-art is a convention that a method named *setVariable* will modify *variable*. If a developer does not follow this convention, then a user of the Class may well conclude that *variable* is not modified.
- Is *variable* modified indirectly? Even if a setter method exists, it is still not clear if other methods also modify the value.

These questions can be answered only by good JavaDocs or reading the code, and are not verified by the compiler.

B. References

At least the private/getter/setter approach provides a mechanism to control the modification of variables. References to object instances represent a more serious problem.

Once anyone obtains a reference to an instance, they can modify any (non *final*) visible variable or call any method defined by the object – including any method which mutates the instance. Even if a developer defines all mutating methods in an inherited Class, and returns references only to a base read-only form of the instance, anyone is free to (cast) the reference to the mutable form

The only safe mechanism is creating a (deep) clone of the original object and returning a reference to that clone. This is cumbersome at a minimum, and if large graphs of interconnected instances are involved can become prohibitively expensive at runtime

C. Example Class

As an example assume a somewhat contrived (and incomplete) message-passing application consisting of a **Message** Class, a static **sendMessage** method, and variables to maintain state. See below. This is a very simple Class but is **unsafe** in all but the most trusted environments.

Note that *Date* is used as a simple, well known (though deprecated) example of a mutable, unsynchronized Class.

```
public class Message implements Cloneable{
  /****************************************************/
  /* Message instance variables and methods           */
  /****************************************************/
  public final Date createdAt     = new Date();
  public final Date sentAt        = new Date(0);
  public final Date ackedAt       = new Date(0);
  public final char[] theMessage;

  public Message(char[] msg)           { theMessage = msg; }
  public boolean hasBeenSent()         { return sentAt.getTime() > 0; }
  public boolean hasBeenAcknowledged(){ return ackedAt.getTime() > 0; }
  public void markSent(Date sent)      { sentAt.setTime(sent.getTime()); }
  public void markAck(Date acked)      { ackedAt.setTime(acked.getTime());}
  /****************************************************/
  /* Static variables and methods to send messages and*/
  /* retain history about the last N messages sent.   */
  /****************************************************/
  public static final int maxSentMessages = 24;
    /** When the Message application started */
  public static final Date systemStart = new Date();
    /** The timestamp of the last message sent */
  public static final Date lastSent = new Date(0);
    /** The most recently sent messages, up to maxSentMessages */
  public static final List<Message> sent = new ArrayList<>();
    /** Messages waiting to be sent */
  public static final List<Message> pending = new ArrayList<>();
  public static void sendMessage(Message msg){
    // .... logic to send the message
  }
}
```

D. Message Class Issues

The basic problem is that all of the variables are *public*. The *final* keyword allows us to specify that the variable itself cannot be changed – but referenced object instances are open to modification, either accidental or deliberate.

For example, the constant showing how many messages will be retained in the *sent* list is safe only because it is *final* and set to a primitive value (which is immutable):

```
public static final int maxSentMessages = 24;
```

However, the timestamp when the last message was sent is specified as:

```
public static final Date lastSent = new Date(0);
```

which says that lastSent will always reference the same Date instance, but anyone, inside or outside of the Class, can change the time: Message.lastSent.setTime(1234)

In order to prevent this, the developer of the Message system must code defensively, hide the object, and return only a copy of the instance to any inquiry:

```
private static final Date lastSent = new Date(0);
public static Date getLastSent() {
  return (Date)lastSent.clone();
}
```

which is safe but has several drawbacks: a) requires more code, b) increases the complexity of the logic, c) allocates a new instance every time invoked, and d) requires documentation about the meaning of the value returned. In particular, the *caller* must realize that the Date returned is a copy and will not be updated if another Message is sent.

If it were desired to provide the *sent* or *pending* lists to a caller, this would be exacerbated. New *ArrayList* instances would be created, a deep-clone copy of each Message copied over, and then the ArrayList returned to the caller.

Finally, the *Message* constructor should make a copy of the passed array. [*String* would be a more natural type for this parameter, but *char[]* is used for the purpose of some of the examples.]

Code complexity and overhead have increased significantly, and also note the problem with the *sentAt* and *ackedAt* variables. Since cloned copies of the *Date* instances are returned to the caller in the cloned Message, when it is sent or acknowledged this will not be reflected in the returned copy – so the caller must re-query and get a new list to see if any *Message* has been updated since the previous query. (Ignoring that some *Message* may also have been dropped from the list.)

Overall, a safe implementation of this trivial Message subsystem will roughly **double** the required lines of code and impose a significant runtime overhead.

Users of *Message* will have a steeper learning curve to understand all of the getter/setter methods, and the semantics of returned objects. Several maintenance programmers in the future must read and understand twice the lines of code in order to safely modify the logic.

None of this adds any value to the application.

III PROPOSED SOLUTION

We need a mechanism to *specify and control* when *variables* and *object instances* can be modified, or more importantly can be known to be *immutable.* This mechanism must be easy for the developer to write, for future users and maintainers to comprehend, and be enforced primarily at compile time.

A series of simple *tags* is proposed which can be declared with *variables* or *types* and control modification of the variable or the ***referenced*** object instance.

E. Ownership

A simple view of the *owner* of a variable or object instance is defined:

- A *variable* is owned by the context in which it is declared. The same as if declared *private* in a Class [JLS sec 6.6.1], or the innermost enclosing context in any other situation (i.e. a *method* or *syntactic block*).
- An *object instance* is (initially) owned by the context in which it is instantiated. There are a few situations whereby ownership may be transferred.

Every other context is considered to be an *outsider* in the following discussion. (There have been other proposals for the definition and handling of *ownership* – see [10])

F. Tags – R (Read-only) or W (Writable)

These specify whether something is *mutable* or *immutable,* and also distinguish between who may modify it – the *owner* or *outsiders*. [Or *I (immutable)* and *M (Mutable)* but R and W appear to be obvious to all developers.]

This will be indicated by one or more character *tags,* which may be appended to:

- Variable definitions
- The types of instance references
- Generics
- Method definitions
- Method parameters
- Method return types
- Constructors
- Classes

In general, these positions are defined as (with some exceptions discussed below):

- Tag 1 – R = may **not** be changed by an *outsider*, W = **may** be changed by an *outsider*.
- Tag 2 – R = may **not** be changed by the *owner*, W = **may** be changed by the *owner*.

These are shown in upper case in this document so that they stand out clearly, and upper case (at least for W) is recommended in source code for the same reason, but the formal syntax is case-insensitive. A *colon* is used as a separator before these tags.

The default case is always *writable* (**W**), so existing (untagged) source code will compile and have the same execution semantics as current Java.

Additional tags are introduced below, and some tags may be inferred during compilation.

1) Variable Definitions

For a variable, a 2-character tag is appended to the *variable name*.

Taking this original statement as an example (simplified by dropping *final* and *static*):

```
public Date lastSent = new Date();
```

and using the new positional declarations, then we could have the following possibilities:

```
public Date lastSent:WW = new Date();
```

Anyone can do anything to this variable (and the target Date instance). The same as the current semantics for *public* (or the default of having *no* tags specified).

```
public Date lastSent:RW = new Date();
```

Outsiders cannot assign to *lastSent* but anyone can change the **value** of the target Date, and the *owner Class* may assign a new instance (or null) to *lastSent*.

```
public Date lastSent:WR = new Date();
```

Outsiders may assign a new value to *lastSent,* but the *owner* may not. Anyone can change the value of the Date. [Admittedly, a somewhat strange semantics for a variable.]

```
public Date lastSent:RR = new Date();
```

Equivalent to *final* – no one may change the value of *lastSent*. The semantics are the same as *final* in that a value must be assigned before exit from the constructor or at initialization if *static.* Anyone may still modify the value of Date.

This provides the beginning of some declarative control, and it is immediately clear to any reader *if* the variable may be changed and *who* may change the variable – but in all cases the *target* instance is completely open to change by anyone (*lastSent.setTime(12345)*).

Also note that the form:

```
public Date lastSent:RW = new Date(0);
```

is logically equivalent to the common pattern where the variable is private but a *getter* and *setter* are defined:

```
private Date lastSent = new Date(0);
public Date getLastSent(){ return lastSent; }
public void setLastSent(Date date){lastSent=date;}
```

So the tags remove the source lines for the getter, and indicate the usage of the variable at the point of declaration. Execution time is usually the same since optimizing compilers will reduce this to a simple variable fetch or assignment.

Some *setter* method is still required, and is valid since the *owner* executes the setter logic and may modify the variable. This method may also handle validation of the new value, and possibly update other related variables

Also note that this is an **unsafe** implementation since the **reference** returned by *getLastSent* allows the caller to modify the (private) Date instance, and the *setLastSent* method allows the caller to retain a reference to the Date passed and also change the value at a later time. In both cases, the Date instance should be cloned.

Finally, all of these tags are *optional* – the developer is free to make a variable *public*, using tags to control access, or continue to declare it as *private* with a *getter* method and (if needed) a *setter* method. Using public/tags does erode encapsulation, and changes the API of the Class. Likewise, a later decision that a public variable needs to be changed to *private* with a getter/setter will also change the API and may be a breaking change. The *SyncProperty* annotation discussed under *Synchronization Tags* may alleviate this problem.

It is expected that most non-trivial Classes will use a mix – with some simpler variables declared as *public* with tags, and other variables as *private*. If some future version of Java supports the *uniform access principle* [11] or *SyncProperty*, so that access to either a variable or a method appears the same to a caller, then the risk of a breaking change to the API would be reduced or eliminated.

2) Variable Assignment Rules

When a variable is assigned to another variable, the tags are independent and may be set as required by the developer.

For example, the systemStart and lastSent variables in the Message system would normally be defined as:

```
public static Date systemStart:RR = new Date();
public static Date lastSent:RR    = new Date();
```

to indicate, and assure, that both of these will always reference the same target instance (the same as *final*).

Now, if you define a new Class you could have this:

```
public class MyClass {
  public Date myStart:WW = Message.systemStart;
  public Date myLast:WW  = Message.lastSent;
        .  .  .  .
```

which allows anyone to change the object instance assigned to *myStart* or *myLast* (or set them to null). More likely, these tags would be set as RW (no outsider may reassign) or RR (to assure that these variables always reference the same instance within MyClass as in Message):

```
public class MyClass {
  public Date myStart:RR = Message.systemStart;
  public Date myLast:RR  = Message.lastSent;
```

```
        .  .  .  .
```

3) References

Any *instance* of an Object is always accessed through a *reference.* This is usually visible as in the statement:

```
        public final Date createdAt = new Date();
```

where the variable *createdAt* contains a reference to the Date instance. Even in more obscure cases such as:

```
        public String formatted = new Date().toString();
```

a reference (visible only to the compiler) will be created, assigned the *new Date()* instance, used to execute the *toString* method, and then abandoned.

A reference always has a *target (object) instance* (or may be null), and several different references to the same target instance may exist simultaneously.

The *initial owner* of any target *instance* is defined as the *first* variable to which a reference is assigned at instantiation [i.e. the result of the *constructor* of the instance].

We do not say that an ***instance*** is mutable or immutable, rather we control whether an instance may or may not be changed through a specific ***reference****.* And note that while an instance may be immutable when accessed through a given reference, there may be other references for which that same instance is mutable (unless declared *Pure*).

4) Type Tag Specifications

The *type specification* associated with a reference variable may also have a 2-character tag, with the same semantics as for a variable. When a *type* is tagged *immutable* (for either the owner or outsiders), then for the **target** *object instance* (in the corresponding owner or outsider context)*:*

- All accessible (visible) variables are treated as immutable (RR).
- Any object instance referenced through a variable is immutable,
- Only immutable *methods* may be invoked.

If a type is tagged as *mutable* then the tag declarations of variables and types within the *target instance* may still control access.

Hence, combined with the tags on the *variable,* we might have the following (partial) list of possibilities (again dropping *final* and *static):*

```
        public Date:WW lastSent:WW = new Date();
```

Which is the same as just *public* or **no** tags – anyone may change either the value of the *lastSent* variable (i.e. which Date instance is referenced or null), or call a method of Date (e.g. *lastSent.setTime(…))* which changes the semantic value.

```
        public Date:RW lastSent:RW = new Date();
```

Where *outsiders* cannot change which Date instance is referenced, nor change the semantic value of the referenced instance, but the *owner logic* may.

This is an important specification – any modification may only be performed by the logic within the owning Class. And this is clear to any reader at the point of declaration.

```
        public Date:RW lastSent:RR = new Date();
```

Where neither *outsiders* nor the *owner* can change which Date instance is referenced. The *owner logic* may change the semantic value of that Date.

The best definition of *systemStart* would be:

```
public Date:RR systemStart:RR = new Date();
```

where both the variable and *Date instance* are *immutable* – no one may change either which instance of Date is referenced nor change the semantic value of Date.

By the *Type Assignment Rules*, no other reference to *the systemStart Date* instance with any *W* tag can be created, so this instance is *immutable* throughout the application and is termed a *Pure* instance.

The natural declaration of the *lastSent* variable would be:

```
public Date:RW lastSent:RR = new Date(0);
```

Which implies that the *owning* Class may update the semantic value of the *lastSent* Date, but that *lastSent* will always reference the same Date instance.

This is important because any outside caller knows that they may safely retain a copy of this reference if they need to track the timestamp of the last message sent. [In a high-volume application, this may well save millions of method calls and cloned Date instances. Note that we are ignoring concurrency issues here.]

The RR tags are also permitted on *primitive types* for completeness, but are redundant since all Java primitives are inherently immutable. So you might have:

```
public static int:RR maxSentMessages:RR = 24;
```

5) Pure Instances

When an object is *instantiated* and assigned to the initial *owner* reference, it may also be designated with a *P* (pure) tag instead of RR for the *type.* This requires that the *constructor* itself is read-only (see Constructors). (*P* may also be used instead of *RR* for a variable.)

This implies that this instance is *RR* when referenced via the owner variable, but the important point is that by the assignment rules we are assured that **no** mutable reference can ever exist for this instance. The *Pure* tag may also be preserved on assignments to other variables, maintaining the fact that this instance is immutable throughout the application.

So now the best declaration for *systemStart* would be:

```
public Date:P systemStart:P = new Date();
```

This statement clearly reveals the intent of the developer and conveys to any reader, the compiler, and any future maintenance developer the important information that:

- The systemStart variable will always refer to the same Date instance,
- The Date instance referenced by *systemStart* is immutable throughout the application.

Note that some Classes – such as String and LocalDate – are Pure by design, no mutating methods exist. The *Pure* tag, however, is more flexible in that it allows specific instances of a normally mutable Class to be declared immutable.

Instances may also be determined to be pure at runtime – see *Pure at Runtime*. This is the only situation where immutability cannot be enforced during compilation.

6) Type Assignment Rules

When a new instance is assigned to a *variable*, the tags on the receiving *type* (target) must be *consistent* with the source:

Source	Target	Notes
WW	WW, WR, RW, RR	Allows passing of the *owner* tag as writable, or essentially duplicating ownership to another context. Either tag may be upgraded to R.
RR	RR	Must be preserved. Cannot be upgraded to P since it is not known if any mutable reference exists.
P	P, RR	P may be downgraded to an RR.
Within the **same** context (owner or outsider)		
WR	WR, RR	Can upgrade the W to an R.
RW	RR, RW	Can also upgrade the W to R.
From source of *owner* to target of *outsider*		
WR	WW, WR, RR	W for outsider *may* be duplicated to owner within this new context.
RW	RR	Outsider R **must** be extended.
From source of *outsider* to target of *owner*		
WR	RR	Owner R must be extended
RW	WW, RW, RR	W for owner may be duplicated

Table 1

For some examples (ignoring tags on the variables for simplicity):

```
public class MyClass {
  public Date:P  myStartA   = Message.systemStart;
  public Date:RR myStartB   = Message.systemStart;
  public Date:RR myLastA    = Message.lastSent;
  public Date:WW myExampleWR= Assuming Date:WR
```

where *myStartA* preserves the *Pure* tag, but *myStartB* downgrades this to an *RR*. Note that this downgrade is important since *myStartA* is now restricted in that only variables with a *Pure* target instance may be assigned (which is great if it can be supported).

myStartB on the other hand can allow an assignment from a variable with **any** tags for the target type – a *P* can downgrade to an *RR* and any other tags can always upgrade to an RR for access through this reference.

myLastA must be an *RR* since *Message.lastSent* is defined as *Date:RW lastSent*, so the *R* for an *outsider* must be applied to both the *outsider* and *owner* tags in this new context.

Conversely, *myExampleWR* can replicate the *W* tag for the outsider to both the *outsider* and *owner* tags within the *MyClass* context. Note that the *Date:WR* specification for *myExampleWR* does convey important information – the original *owner* of this instance will make no changes to the target and essentially hands it over to outsiders. (This usually occurs when returning an instance from a method.)

If a *P* is preserved, this tells the reader, any future developer, and the compiler, that the target instance is immutable. There may be other references to the same target instance, but every such reference must be at least an *RR.* Preserving a *P* tag allows the compiler to utilize all possible optimizations. For example, a P tag may allow the compiler to bypass synchronization associated with some methods of this Class (see *Pure at Runtime*).

IV DECLARATIONS WITH TAGS

The goal is to allow the specification of R/W tags throughout the type specifications of the language – beyond the simple variable and type declarations discussed previously.

7) Arrays

The declaration of an *array* may include the normal tags after the *type* of the array elements, and also include tags within the brackets to indicate whether new elements may be assigned to slots within the array. For example,

```
        public Date:RW[:RW] dateArray:P = new Date[12];
                or the equivalent
        public Date:RW dateArray:P[:RW] = new Date[12];
```

declare an array where the elements are Date instances which may be changed by the *owner* and where the owner may assign new instances to any of the elements of the array. The *dateArray* variable itself is Pure so it will always reference the same array instance.

If an array has multiple dimensions, then the tags within the brackets control the allowed assignment to each dimension. For example,

```
        public Date:RW[:P][:WW] dateArray:P = {new Date[1], new Date[3], new Date[5]};
```

defines a ragged Date array. The first dimension contains 3 sub-arrays with 1, 3, and 5 slots respectively. Anyone may assign a new element (or null) to any of the sub-arrays, but each sub-array itself will always be the same array instance. Only the *owner* may change any of the target Date instances assigned to the arrays (even if the instance was assigned by an *outsider*).

8) Generics

Tags may also be applied to generic specifications, and recursively to any embedded specifications. So the original definition:

```
        public static final List<Message> sent = new ArrayList<>();
```

should be tagged as:

```
        public static List:RW<Message:RW> sent:P = new ArrayList<>();
```

which specifies that the *List* may not be modified by any outsiders, nor may any *Message* instance contained within the List. However, the *owner* may modify the List itself (add or delete entries), and may also modify any *Message* instance contained within the List (e.g. set the time acknowledged). The actual List *instance* will not change since *sent:P* is specified.

Since there may be multiple readers and the owner may update the list, in a multi-threaded environment it should be synchronized as:

```
        List:RW<Message:RW> sent:P = Collections.synchronizedList(new ArrayList<>);
```

and the *Message* Class should also provide the appropriate synchronization (or use *Synchronization Tags*).

A more complex example could be something like:

```
        public Map:RW<Date:P, List:RW<Message:RW>> history:P = ……
```

Which (presumably) represents a Map, keyed by a Date (with 00:00:00), with a list of the Messages processed on that Date. The *owner* may modify the Map (add a Date), may modify each List (add a Message), or any Message itself (set the time acknowledged). No *outsider* may make any changes at any level of this structure. Also, any Date instance added as a key must be immutable throughout the application (always a good quality for the key of a Map), and the *history* variable will always refer to the same Map.

This essentially allows the developer of the Message system to provide a *safe* view of an internal working data structure to all outsiders. Without R/W tags, to do this safely would require all the complexity discussed above (creating deep clones at every level of the structure) and significant runtime overhead. (Ignoring any synchronization issues in this example.)

9) Declaration of Generics

The compilation of generics requires additional logic to determine if a generic is *R-safe* – that is, it treats all actual instances of the *type parameter(s)* as read-only.

See [8] and [9] for an explanation of how generics are implemented in Java. Note that it is not possible to instantiate a *new* instance of a generic type *T* within the Class, so all code within the generic will be an *outsider* for any instances of type *T*, and only the outsider tag is effective. However, the *owner* tag can also be provided and is important documentation to indicate to a reader if instances may be modified by the original owner.

A trivial *R-safe* generic which returns a timestamped *toString*() might be:

```
public class GenericRSafe<T> {
  public String timeStamped(T obj) {
    return String.format("At: %,d -- %,s", System.currentTimeMillis(),
                                                obj== null ? "(null)" : obj.toString()); }
  }
```

since all of the methods invoked within *timeStamped* are read-only. (toString is known to be read-only since Object.toString:R is read-only, and can be overridden only by a read-only method – see *Method Overriding*).

So this generic may be instantiated by either:

```
new GenericRSafe<MyClass> or new GenericRSafe<MyClass:R>
```

On the other hand, the declaration:

```
public class GenericNotRSafe<T extends Date> {
  public void setToNow(T obj) { obj.setTime(System.currentTimeMillis()); }
}
```

is not *R-safe* since the *setToNow* method mutates the *obj* parameter. This can be instantiated by

```
new GenericNotRSafe<Date>
```

but will generate a compile error on:

```
new GenericNotRSafe<Date:R>
```

[The author believes, but has not verified, that all of the standard Collection Classes are *R-Safe*.]

10) Tagged Generics

Although in general an *R-Safe* generic provides the most flexibility, the *type* specified in a generic may also be tagged:

```
public class GenericWithR<T:R>
```

to require that any instantiation specify an actual type with an R (or P) tag. So:

```
new GenericWithR<Date>
```

will generate a compile error.

The safest declaration for a Map might require that all keys be *Pure*:

```
interface Map<Key:P, Value>
```

11) DownCasts and UpCasts

A variable may be explicitly *upcast* to a more specific type, or (usually implicitly) *downcast* to a less specific type:

```
public Date myDate  = new Date();
public Date myDateA = myDate;         // No cast
public Object obj   = myDate;         // Downcast
public Object obj   = (Object)myDate; // Explicit downcast
public Date myDateB = (Date)obj;      // Explicit upcast
```

While *upcasts* will often appear in source code for specific situations, most *downcast* situations are implicit and are very common as the *parameters* or *return types* of methods. For example, a simple logging system might declare:

```
public void info(Object obj) …
public void error(Object obj) …
```

where **any** type of object may be passed, and the *toString* method invoked to obtain the information to be logged (plus a check for *null*).

Just as in any assignment, R/W tags must be preserved and may be upgraded in either a *downcast* or *upcast.* More importantly, when generics are specified, this rule must be observed at every level of the type structure. For example:

```
public List:RR<Map:RR<Date:P, List:RR<String>> list1 = ….
public List:RR<Map:RR<Date:P, List<String>> list2 = list1  // compile error
```

list2 = list1 fails since the RR on the embedded List:RR<String> is not preserved.

An assignment is termed *ragged* if the generic structure is compressed – for example:

```
public List:RR<Map:RR<Date:P, String>> list1 = ….
public Object objA = list1;                          // Fails
public Object:RR objB = list1;                       // OK (RR preserved)
```

The general rules are that in the case of *ragged* assignments:

downcasts	The highest (most restrictive) R/W level from the source must be preserved by the target
	If all source entries are Pure then may be assigned to Pure (or downgraded to RR)
upcasts	Any source R must be preserved, distributed to all levels of the target generics. Any *owner* or *outsider* tag must be upgraded if crossing to the opposite context
	Any W may be upgraded to an R
	If the source is *Pure* the target may be *Pure*

For example,

```
public List:RR<Date:RR> list1 = ….
public List:RR<Object>  list2 = list1;       // Fails, Date:RR not preserved on Object
public Object           obj1 = list1;        // Fails, no RR tags preserved
public Object:RR        obj2 = list1;        // Succeeds
list1 = (List:RR<Date:RR>) obj1;             // Succeeds
list1 = (List:RR<Date:RR>) obj2;             // Succeeds
```

12) Cumulative Immutability

Immutable status may occur at any step through a chain of references. For example:

```
fooA.fooB.fooC.someVariable = ….
```

will be disallowed if the *type* tag at any step is R. In this example, if *fooB* is defined as:

```
SomeClass:RR fooB;
```

then all references to the right of *fooB* become immutable so the assignment statement will generate a compile error.

V METHODS, CONTEXT, OWNERSHIP TRANSFER

When a method is invoked, there are several important mechanisms involved:

- The method may be declared as immutable and cannot modify any permanent variable,

- Parameters may be declared as immutable,
- Return values may be tagged to indicate mutability within the context of the caller,
- Ownership of an instance may be transferred into, or returned from, the method (in limited contexts).

G. Immutable (Read-Only) Methods

A *method* of a Class is considered *immutable* if during execution:

- No instance or static variable is assigned a new value
- All parameters are treated as immutable,
- All internally referenced object instances are treated as immutable,
- Any referenced instance or parameter will be passed to any other method as an immutable parameter,
- The method does not return a reference to *this*, or any instance or static object, as a mutable reference.

Note that an immutable method may still return different values on successive calls. For example,

```
System.currentTimeMillis is read-only (R? below).
```

H. Method Tags

A *method* is declared to be mutable/immutable by appending one or more *compatible* tags to the method name with one of these values.

Tag	Semantics
W or no tag	Method is a *mutable* method (default).
R	Method is *immutable* as defined above.
C	Method is *immutable* during *construction*.
T	Modifies only *transient* variables
L	Modifies only *instance* variables.
?	All other tags should be accepted by the compiler, but this is *forced*.

Table 2

A *W* method may change the state of this instance (if an instance method), a parameter (if so tagged), a mutable referenced instance, or a global variable. This is the implied semantics of all methods currently (and the default if no tags).

The *C* tag may be applied to a *constructor* (although *R* is preferred stylistically in order to be consistent with other methods). It may also be applied to any other method within the Class (for example, a private method invoked by multiple constructors to perform common initializations). The difference is that any *constructor* or *C-tagged* method may modify *instance variables* during the *construction phase* of an instance. A *C-tagged* method is considered a *mutable* method if invoked outside of the construction of an instance.

A *T* tag restricts modifications to *transient* instance variables, target instances referenced by transient variables, or any globally visible transient variables or instances. For example, some objects compute and save a *hashcode* in a transient variable. T may be combined with an R tag.

Note that this is a major expansion of the usage of the *transient* keyword, and (if with R) differentiates between changing the *semantic value* of an instance versus incidental changes (such as computing and saving a hashcode). Presence of a method with a *T* tag allows treating this method as an immutable method, but may preclude treatment of any instance as *Pure* since cache coherence mechanisms may be bypassed when in fact a transient variable has been modified. See Future Research.

The *L* tag is a restriction indicating that this method accesses (or modifies) only instance variables. This is provided to enable possible performance improvements (especially in multi-core contexts) and as a design notice for future maintenance cycles.

The *?* tag is provided to handle those cases in which the developer declares that the *semantic state* of the instance meets all the requirements of the other tags, but that some of the formal rules must be violated. For example, issuing a log message (calling a mutable method), or the accumulation of profiling statistics.

If tagged as *?* the compiler will issue warnings for all cases which would normally result in a compile-time error.

1) Compatability Rules

L, T, and ? are all narrowing and can be used with any other tag (although ? with W is meaningless). W with R is an obvious contradiction. R with C is allowed but redundant since R is more restrictive.

JNI native methods may also be tagged but must include ? if any tag beyond W appears. This is entirely at the direction of the developer and the compiler can offer no further analysis.

For example, using the sample *Message* Class above:

```
public boolean hasBeenSent:R()……
public void markAsSent:W(Date timeSent)……
```

specifies that the *hasBeenSent* method will not change the state of this *Message* instance (or any global variable or instance) but the *markAsSent* method may.

I. Method Parameters

On a method call, the *caller* is considered as the *owner* of all parameters, and the method (the logic within the method) as the *outsider* in terms of tag positions.

For each parameter, tags may be specified for both the *type* and the *variable* of the parameter. For example:

```
public void markAsSent:W(Date:RR timeSent:RR)
```

The initial R on *timeSent* assures that no assignment is allowed within the method, which can serve as documentation for the developer and an internal consistency check. [The *owner* tag is meaningless since all parameters are passed by value, so the caller technically does not have access to the variable within the method. The formal syntax allows it for completeness.]

The first tag on *Date* indicates that this instance is immutable by the *outsider* (method logic), and may not be passed to any other method unless through an immutable parameter – so the *caller* is assured that the passed instance will not be modified within this method. [Note that if this method itself is W, it would be possible to change a parameter marked R if there is a globally visible mutable reference to the *same* instance, or if the same instance is also passed to another parameter as mutable.]

The second tag is ineffective since the method logic is always considered to be an outsider, but can serve as documentation that the instance may be modified by the *owner (caller)* during the execution of this method (e.g. by another thread). Note that since the method is considered the outsider, if the passed reference is assigned to any other variable the owner tag must be upgraded to an R by the assignment rules.

A pure (*P*) may also be specified and requires that any actual argument passed must be Pure.

So the shorthand in this case could be:

```
public void markAsSent:W(Date:P timeSent:R)….
```

which is a method which may mutate **this** instance, but will not modify the instance referenced by *timeSent*, will also not change the value of *timeSent* itself, and the caller guarantees that the value of the Date will never be changed (which is an over-specification in this case).

If the parameter is defined as mutable, the method may need to make a defensive copy if it intends to hold a reference beyond the scope of the method call. For example, the *Message* constructor in the original example should make a copy of the passed character array unless it is marked as *Pure.*

If the method itself is marked as *R*, then any *untagged* parameter type is inferred to be R and a parameter explicitly tagged as W is an error.

J. Method Overloading

The *tags* of the parameter *types* are considered part of the *signature* of the method when determining overrides and overloading. For example, two constructors could be declared as follows:

```
public Message:R(char[] msg:R)
public Message:R(char[]:P msg:R)
```

where the first constructor (with implied WW tags for *char*) needs a defensive copy of the character array, while the second form could safely store a reference.

For *overloading* the compiler will always favor the method with the most restrictive type tags which can be applied, considering parameters from left to right. So given these definitions:

```
public void someMethod(Date param1, int param2)
public void someMethod(Date:P param,int param2)
```

the second method will be chosen any time the *actual* parameter at the call site is Pure.

For *overloading* it is an error to have tags on the method itself as the only difference in the method signature.

K. Method Overriding

For an *override,* a method with an R tag can override a method with a W, but not the other way around. If an overriding method specifies **no** tag at all, then it is inferred to be the same as the method which is overridden (if overriding an R, then the current method must validate as R).

As an example, consider:

```
In Foo: public int someMethod:W(…..)
In FooA extends Foo: public int someMethod:R(…..)

Foo myVar = new FooA(….)
```

myVar is declared as Foo (with the mutable method), and will be treated as such during compilation. During execution, *myVar* actually contains a reference to an instance of FooA (with the immutable version of the method) so invoking *myVar.someMethod(….)* can do no harm. Conversely, overriding some *method:R by method:W* would erroneously execute a mutable method.

Note that a call to *super.someMethod* is a compile-time error within *FooA.someMethod* since an immutable method would be invoking a mutable method.

Within Object, the *clone, equals, getClass, hashCode,* and *toString* methods should be flagged as R. It may also be possible to tag the *notify…* and *wait…* methods as R or R?.

Note that this is a **breaking change** in many cases. For example, in recent Java versions *String* caches the value of the hashcode in the *hash* and *hashIsZero* fields – these must be changed to be declared as *transient* and the *hashcode()* method tagged as *hashcode:RT()*. The *toString:R()* method may also lead to many compile-time errors for overriding Classes in which *toString* has not followed the expected convention and is not a read-only method. Many of these may be correctable by defining some fields as *transient* and then declaring *toString:RT()*.

An alternative would be to declare an intermediate:

```
public class ObjectTags() extends Object {
  public Object clone():R{ … }
  public boolean equals(Object obj):R{ … }
…..
```

which overrides all of these methods as read-only. Then new code Classes could extend *ObjectTags*. While not a breaking change to existing code, this also introduces additional complexity into the language. In the worst case, this approach could lead to distinct tagged and untagged versions for many libraries.

L. Return Types

When a method returns a value, the *method* is considered the *owner* and the *caller* the *outsider.* The return value *type* may be tagged in the usual way.

For example, if it is decided to hide the *lastSent* variable, then this implementation can be specified:

```
private static final Date lastSent = new Date(0);
public Date:RW getLastSent:R() {return lastSent;}
```

which specifies that the *outsider* (*caller*) may not modify the object instance, but that the *owner* (*method)* still retains a reference to this instance and may change it. [This may be the single most valuable feature of this whole approach – the owner of an instance can return references and be assured that no outsider will modify that instance.]

If assigned to a local variable, then the owner tag must also be upgraded to an R:

```
public class MyClass {
  private Date:RR lastSent=Message.getLastSent();
       ….
}
```

Alternatively, if a defensive copy is created and returned then this could be specified as:

```
private static final Date lastSent = new Date(0);
public Date:WR getLastSent() {
  return (Date)lastSent.clone();
}
```

which tells the caller, essentially, that this is a clone which they may modify as desired. [See Transfer of Ownership for a stronger approach.]

This can be even more important when a collection is declared as *immutable*:

```
List:P<Message:P> someMethod(….)
```

which specifies both that the *list* is immutable, but even more importantly that all of the object instances contained within the list are immutable.

For example, consider a hypothetical method which retrieves a list of messages sent to a given IP address. If declared as:

```
public List:RW<Message:RW> getMessagesSentTo:R(IPAddress:P ip:R);
```

then the returned list may be changed by the owning Class after being returned to the caller. The implication is that the method has simply returned a reference to an internal working list.

If the method has actually created a copy of the working list (and a deep clone of each Message), then this could be:

```
public List:WR<Message:WR> getMessagesSentTo:R(IPAddress:R ip);
```

which implies that the *caller (outsider)* is free to make changes, but that the *method* has not retained any mutable references. (But note that this is not strictly enforced by the compiler!)

The *caller* may receive this as a WW since this is an assignment from the owner to outsider:

```
public List:WW<Message:WW> someVar = getMessagesSentTo(IPAddress ip);
```

If only a shallow clone has been utilized for the Message, then the correct declaration is:

```
public List:WR<Message:RW>    getMessagesSentTo:R(IPAddress:R ip);
```

which indicates that the caller may modify the List if desired, but that the Messages within the list cannot be modified by the caller. However, the Messages may still be modified by the owner. So this assignment, with the required context change upgrade of RW -> RR, would be:

```
public List:RW<Message:RR> someVar = getMessagesSentTo(IPAddress ip);
```

M. Assignment or Return of a Parameter

Any input parameter which is assigned to a variable within the method obeys the normal assignment rules above. In particular, if the declaration is:

```
public void markAsSent:W(Date:R timeSent:R)
```

then an internal assignment such as:

```
Date:RR localVariable = timeSent
```

must promote the implied *owner* tag to an *R* since it is in a different ownership context. (This may be inferred by the compiler in order to compile untagged source.)

Any parameter which is *returned* is treated as an implicit assignment so must also be promoted to an RR if returning any input parameter tagged as RW.

N. Transfer of Ownership

In some important contexts, a caller may pass *ownership* of an instance into a method, or a method may export the *ownership* of a returned instance to the caller.

In either of these cases, a *T (transfer)* tag is the single tag for the *type*.

For example, the method to send a message could be:

```
public static void sendMessage:W(Message:T msg:R)
```

The *T* tag requires that from the context of the *caller*:

- The caller is the current owner of this instance,
- No other references exist.

The compiler must generate an assignment of **null** to the calling variable (unless it will immediately be out of scope). Within the receiving method, the actual argument is taken as a WW. Also note that the *actual argument* may well not be visible, for example in:

```
sendMessage(new Message(…));
```

the compiler creates a hidden variable for *new Message* which is immediately out of scope.

If the caller would like to have a reference to the Message, then *sendMessage* might be declared as:

```
public static Message:RW sendMessage:W(Message:T  msg:R)
```

so the caller may invoke this as:

```
public Message:RR myMessage = sendMessage(new Message(“My Message”.toCharArray())
```

where the returned *RW* is promoted to *RR* as required by the rules of assignment. (It is also possible to return **Message:WW** since *new Message* defaults to WW. This is usually not advisable, but might be useful during the transition of a codebase.)

If ownership is transferred upon *return* from a method, then the same rules as above apply to the logic within the method. Transfer of ownership can be especially useful if a *factory method* is defined which instantiates and returns a complex instance. For example, if a factory method is defined which accepts a *String* parameter, converts to a *char[]*, and instantiates a Message, this would be declared as:

```
public static Message:T messageFactory:R(String:P  msg:R)
```

and the caller is assured that no other references exist after the return from *messageFactory.*

O. Scope Implications

When a method or local scope within {..} brackets exits, any local variables cease to exist. This allows utilization of mutable variables internally, but still return immutable values to the caller.

For example, assuming that *deepClone* and *isToIP* methods exist for the Message object, then the *getMessagesTo* method might be implemented as:

```
public List:P<Message:P> getMessagesTo:R(IPAddress:P ip:R) {
  List<Message> rslt = new ArrayList<Message>();
  for(int i = 0; i<sent.size(); i++)
    if(sent.get(i).isToIP(ip))
      rslt.add((Message)sent.get(i).deepClone());
  return rslt;
}
```

whereby the *rslt* List (and the cloned Message instances) are mutable within the method, but can be returned as pure since no other references exist upon exit (or could be *WW* to allow the caller to manipulate).

P. Lambdas

The parameters (and variables) of lambda expressions may be typed exactly like method parameters if *explicitly* typed. The tags for the *return* value may be appended after the closing parenthesis of the declaration. For example,

```
(Date:R startDate, Date:R endDate):P -> { return ….. }
```

is a lambda which takes two read-only parameters and returns a Pure value.

In order to adapt to common usage, in an *implicit* lambda expression the *type tags* may be appended to the variable name:

```
(startDate:R, endDate:R):P -> { return ….. }
```

indicates that both *startDate* and *endDate* will not modify the (implied) Date instance passed.

Q. Constructors

A *constructor* can be tagged like any other *method* but with some critical differences:

- modification of any *instance* variable is not considered a mutation,
- the *type tags* normally associated with a return value may be appended after the method tag.

All of the normal assignment rules then apply, so for example the *Message* constructor could be declared as:

```
public Message:R(char[]:P msg:R)
```

(or *Message:C*) to indicate that executing the constructor has no mutable side-effects beyond the member variables.

As a stronger declaration, it could also be:

```
public Message:R:P(char[]:P msg:R)
```

which states that Message is an immutable constructor, but also that the *result* (the constructed instance) may be treated as *Pure.* This would not work in our example, since the logic must update the Message to indicate the time when it was sent and when it was acknowledged, but does allow a *Class* to specify that all instances are immutable upon instantiation with this constructor. It would also be possible to define alternate constructors which specify different return tags which allow mutability, or create a factory method to encapsulate all instantiations.

R. Right Hand Side (RHS) Tags

A weakness of the current *final* concept is that a logical contract is implied with any user of the variable – this variable will not be modified – but the *final* keyword may be removed in the future without notice.

One of the examples above had this:

```
public class MyClass {
  public Date:RR myStart:RR = Message.systemStart;
  public Date:RW myLast:RR  = Message.lastSent;
}
```

which has semantic dependencies on the sources of the assignment. The developer of *MyClass* expects that *Message.systemStart* (and the referenced Date) will not be modified, and that *Message.lastSent* will always refer to the same instance of Date, and that this Date instance will be updated to reflect the time when the last message was sent. If the *Message* Class is modified in the future to use a different mechanism to record the system start time or the last sent time, then this implied contract is broken – again with no notice to any referencing Classes.

In the extreme *Message.systemStart* and *Message.lastSent* could be changed to *Date:WW systemStart:WW* and *Date:WW lastSent:WW*. and the assignment would still be legal since a W may always be promoted to an R.

This is addressed by allowing *tags* on the right-hand side of an assignment to specify either:

- The *minimum* requirement for the source tags, or
- An exact match to the tags on the source (with an **X** before the tag)

Two sets of tags may be specified – the first for the source *variable* and the second the *type*.

So the safe definition of *MyClass* would be:

```
public class MyClass {
  public Date:RR myStart:RR = Message.systemStart:RR:RR;
  public Date:RW myLast:RR =  Message.lastSent:RR:XRXW;
   }
```

which specifies that *Message.systemStart* must be declared, as a *minimum*, as *Date:RR systemStart:RR*. A *Pure* in place of the *type:RR* is also legal and would be preferred in this case to assure that *no* mutable reference exists.

The RHS declaration for *Message.lastSent* specifies that the variable must always refer to the same Date instance (*RR or P)*, and that the Date instance must be declared as *:RW* (exactly) – implying that the Date will be updated to always reflect the time of the last message.

This ensures that if the semantics of the *Message* Class have been modified, at least a compile error occurs to notify the developer of *MyClass*. This can be especially important when using third-party libraries where the source code may not be available to the *MyClass* developer.

1) Method Calls

In the same vein, tagging a method as read-only is an implied contract. Tags may be specified at the point of call to verify this contract:

```
myVar = referenceVar.someMethod:R(param1, …)
```

will generated a compile error if *someMethod* is not tagged as read-only.

Again, this is most useful if the developer of the application does not have control over the library which provides *someMethod*, but can also be useful within any large application to ensure that a future maintenance cycle does not inadvertently violate an assumed contract.

At the very least, this provides documentation at the point of call that *someMethod* is expected to be read-only.

In all such contexts, even though annoying that the expected contract with a library has been changed, it is still preferable to fail at compile time instead of introducing runtime errors.

Note: These RHS tags are effective during compilation, but will not detect a problem if a new Jar with the updated library is introduced to the classpath without recompilation. See Future Research – Enhanced ClassLoader.

VI CLASS AND INTERFACE TAGS

The *Class* or *Interface* may also be tagged in order to specify the *minimum* level of tags for:

- Static variables
- Static variable types
- Static methods

- Static method return types (outermost level only if a generic)
- Instance variables
- Instance variable types
- Instance methods
- Instance method return types (outermost level only if a generic)
- Constructors (not present for an Interface)
- Constructor type tag (not present for an Interface)

This is an optional (and somewhat clumsy) construct but can be used to:

- Allow the designer of the Class to establish some basic mutability requirements,
- Allow the compiler to verify that there are no violations, and
- Provide the reader with an overview of the mutability of the Class..

[There is nothing worse than the JavaDoc stating that a Class is immutable, and then find that some maintenance cycle has introduced a *setter* method!]

Since this consists of a large number of settings, the syntax adapts the [..] of arrays (two), with comma-delimited elements for each of the items above, in the order as given (with the first array for static settings, and the second for instances plus constructors).

```
public class MyClass[RW,RW,R,RW] [RW,RW,R,RW,R,RR]
```

which requires, for example, that *all* static variables in the Class must have tags of RW (the stated minimum), RR, or P. The same applies for each of the other positions.

Unspecified elements can be indicated by two adjacent commas and will default to W or WW. If the entire bracket section is omitted or empty, then everything defaults to W.

At any point in the sequence, specifying * instead of tags will replicate the preceding value to all further elements to the right:

```
1) public class MyClass
2) public class MyClass[WW,*][WW,*]
3) public class MyClass[P,*] [P,*]
4) public class MyClass[WW,WW,W,WW] [P,*]
5) public class MyClass[P,*] [WW,*]
```

where (1) is the default of no specification, defaulting to all Ws, (2) is an explicit specification of all W's, (3) specifies a Class where all static variables/methods and all instances are Pure, (4) specifies that static variables/methods are mutable but all instances are Pure, and (5) is the opposite in that all static variables/methods are Pure, but instances are mutable.

It would be possible to have the Class-level tags be substituted for any variable/method for which **no** tags are specified (instead of a compiler error). The downside is that if someone is reading the code it would not be apparent that, say, a method is in fact read-only even though no tag appears. If viewed through a good IDE, however, this situation could be handled to show the inferred tags (highlighted by italics, coloring, or some other indication). This might be an excellent way to convert untagged code while requiring minimum changes to the source.

For example, in the *java.time* package, LocalDateTime, LocalDate, etc would all be declared immutable by tags at the Class level (instead of in the JavaDoc), and this would be guaranteed by the compiler.

```
public class LocalDateTime [P,*] [P, *] { … }
```

VII PURE AT RUNTIME

An instance created as *Pure* cannot then be modified.

For many common data structures, for example trees or lists, this is not a problem – the *leaf* nodes can be instantiated as Pure, and then the next level/link instantiated to refer to these. For example:

```
public class FooList<Date:P> {
  public FooList:P<Date:P> next:P;
  public Date:P date:P;
 /** Create a new node on the front of the list */
  public FooList:R(Date:P date,FooList:P<Date:P> next){
    this.date = date;
    this.next = next;
  }
  /** Create the initial node of the list */
  public FooList:R:P(Date:P date){
    this(date, null);
  }
}
```

However, if 2 (or more) instances hold references to each other, then neither can be declared to be *Pure* at compile time. For example,

```
public class FooA{
  private FooB:RR refFooB;
  public setFooB(FooB:RR fooB){
    this.refFooB = fooB;
  }
}
public class FooB{
  private FooA:RR refFooA;
  public setFooA(FooA:RR fooA){
    this.refFooA = fooA;
  }
}
```

won't work since both must be instantiated and then the **setFooA/setFooB** methods must be invoked to set the cross-reference.

This issue extends to the general case of any group of objects which have references to other objects within the group, but which are otherwise immutable after initial construction. For example, a doubly-linked List, or some tree structures in which each node has a list of children but also a reference to its own parent. The situation might also occur if using an untagged library. For example, if a library provides a method to read a page from the Web, parse the HTML, and return the root of the parse tree, then it might be called as:

```
HTMLNode:RR rootNode = retrievePage( <URL> );
```

and the *HTMLNode:RR* declaration implies that all of the objects in the parse tree are in fact *Pure* since no mutable reference exists (assuming that *retrievePage* has not leaked a reference).

The objects assigned as Pure at initial instantiation are termed *Pure at Birth*. The second set of cross-referencing objects cannot be pure at birth, but may be determined to be *Pure at Runtime* (though this may be an expensive operation).

In order to determine if an object can be considered Pure, it must be determined that all *existing* references to that instance are RR or Pure (since none of these references may subsequently be assigned to a mutable reference).

This requires logic similar to the Garbage Collector – tracing all references to objects from the root(s). In addition, this trace must refer to the full type information (signature) rather than the type erasure normally used by the JVM. For example, **List<Date:RR>** implies that references stored within the List are read-only to the target Date object.

There are several ways in which this functionality can be implemented and made available to the developer – as methods provided by *java.lang.System* or *java.lang.Runtime.*

S. Pure Single-Instance

This would provide a method of the form:

```
static public Object:P asPure(Object obj);
```

which would trace all live references to *obj* and return **null** if any mutable reference is detected or a reference to *obj* which has been elevated to Pure.

While the simplest to invoke, this method would be expensive.

T. Multiple Instance

This would provide a method of the form:

```
static public Object:P[] asPure(Object[] obj);
```

which would take an array of objects and return a 1-1 array with either a **null** or pure reference.

This also requires a full trace from the root(s) but amortizes the expense over many objects.

U. Enhance Object (or ObjectTags)

As support for Pure objects, the Object Class would be enhanced as follows:

```
public class Object {
 private boolean pure = false
 public boolean isPure() {return pure;}
 public Object:P asPure(){return pure ? this : null;}
        ………. . .
}
```

(where the *pure* variable would probably be implemented as a bit-flag in the object header) and a System method added:

```
static public void determinePure();
```

which will scan **all** reachable objects and set their *pure* flag.

Clearly, enhancing the definition of *Object* is a radical change to the language (and JVM), and may well have insurmountable compatibility problems with existing code, but does add critical immutability capability to the language core, and automatically addresses the pure/mutable status of **all** objects every time *determinePure* is executed. Compatibility issues may be alleviated by using unusual names for the above or add this to the *ObjectTag* Class.

Note that the *determinePure* execution might be designed to execute a *happens-before* protocol just before exit (i.e. synchronize all caches on a multi-processor system). This supports some of the optimizations discussed below.

V. Pure at Runtime Optimizations

Note however the following potential optimizations when determining *Pure at Runtime*:

- Any object which is Pure at Birth is marked as pure upon instantiation,
- As soon as the scan encounters a reference which is RR or Pure, that portion can terminate since all paths through this reference must be immutable.
- Once marked as pure, an object will always be pure since it is not possible to create a mutable reference.
- If Object is enhanced, setting the pure bit may be incorporated into the normal processing of the Garbage Collector (although this adds the complexity of using the full type information (*signature*) rather than the erasure).

It is not yet determined how much the above optimizations will reduce the overhead of runtime determination of immutability, but in a heavily tagged application with a large percentage of RR instances this might be substantial. (In the example above of an HTML parser, all of the nodes in the parse tree would be marked as *Pure* and scans can terminate as soon as the *HTMLRoot* variable is encountered.)

W. Optimizations with Enhanced Object

If Object is enhanced to include the *pure* flag, there may be several possible optimizations in the JVM (or JIT compiler) – these require further research:

- Ignore *synchronize* processing on any *instance* method if pure and the method references only instance variables (and does not have a *T* tag).
- Ignore the memory boundary processing for any *volatile* instance variable (unless transient and any method has a *T* tag). Assumes the cache coherence step is included as defined above.

Both of these may require a *@SkipIfPure* annotation for the *volatile* or *synchronized* keyword to declare that no variables external to the instance are dependent on these actions.

In modern applications with multiple processors and many, many threads running in parallel, reducing the incidents of synchronization (and cache clears) can lead to major performance improvements.

X. Annotations for Tags

It would be possible to use *annotations* to specify tags instead of the proposed *:RW* mechanism.

For simple situations this would be a reasonable approach:

```
@tags(Date=RW, lastSent=RR)
public Date lastSent = new Date();
```

However, for nested declarations it would introduce the danger that a complex structure must be introduced twice and match exactly. For example:

```
public Map:RW<Date:P, List:RW<Message:RW>> history:P = ……

@tags(Map=RW, Date=P, List=RW, history = P)
public Map<Date, List<Message>> history = …
```

would generate no warning that the tags for Message were forgotten, and worse no error if a Message instance within the list is mutated by outsider logic. Also, if the contents of the list were a *Date* rather than a *Message* then the annotation as shown would be ambiguous, so a more complex, nested, annotation structure would be required. The complexity would increase further to handle RHS tags, method tags, and Class/Interface tags.

These considerations led to the rejection of an annotation-based approach in favor of the *:RW* syntax as described.

VIII CONCURRENCY ISSUES

In a multi-threaded environment, the programmer must be aware of concurrency issues when declaring a non-private variable. This will become even more prevalent if *tags* are used to protect a *public* variable instead of making it *private* with synchronized getter and setter methods. The *Message* Class in this document would almost certainly operate as a multi-threaded application.

Y. Synchronization Tags – S, U

If a variable is visible (public, protected, or package visible) with tags, then any requirements for concurrent access to that variable must also be specified. This is supported by defining an ***S*** tag for a *synchronized R (read)*, or a ***U*** tag for a *synchronized W (update)* (and implied synchronized read). These tags may also appear for a *method*, but are redundant with (and will be implemented by) the existing *synchronized* keyword.

Combinations such as *RU* or *SW*, with synchronization for one context but not the other, are an invitation to a race condition. This will generate an error during compilation unless flagged with a *@NoRace* annotation to handle special situations where no race condition can occur.

For example, in the *Message* Class, declarations such as:

```
public Date:SU sentAt:P = new Date(0);
public Date:SU ackAt:P  = new Date(0);
```

would handle synchronization of the target *Date* instance.

The *type tag* assignment rules of Table 1 apply, with the additional constraint that synchronization must be preserved, so if *ref* is a reference to a *Message* with the *Date:SU* tags:

```
public Date:RR myVariable = ref.sentAt;
```

will generate a compile error.

If the target Class is designed for concurrent usage and is synchronized internally, or is immutable by design, such as LocalDate, the *S* and *U* tags are not necessary. This is always the goal, since preserving synchronization can be an awkward API for callers. They are used in the contrived *Message* class so that the unsynchronized *Date* can be used safely.

1) Basic Implementation

Given that *ref* is a reference to a *Message* then:

```
myVariable = ref.sentAt.getTime()
```

would compile to:

```
synchronized(ref){  myVariable = ref.sentAt.getTime(); }
```

For a static variable or method, the *synchronized* keyword would reference the *Class* of the object.

2) Desired Implementation

The *Basic Implementation* will generate code in all of the Class definitions which have *S* or *U* tags. It would be more efficient to generate code in the *target* object, which can be used to support the *S* and *U* tags. This will be controlled by the *SyncProperty* setting (as either a compiler flag or annotation). This will default to *off* initially, but may at some point in the future default to *on*. If *on*, then the following will be active.

Any *method* in the target which is not already marked as *synchronized* would generate a shadow, synchronized method which would invoke the original method. For example, the *getTime* method in *Date* would also compile this shadow method:

```
public synchronized long getTime_S$() = { return getTime();}
```

which would then be invoked from any calling location with an *S* or *U* tag in effect.

Any declared *variable* will be compiled as a *property* [12 pg 80], which will hide the variable as *private* and create *getter* and *setter* methods with the name of the original variable.

```
For: public MyClass:XX myVar = ……
```

where XX indicates any set of tags (or nothing) on the type, a full compilation would be:

```
private MyClass:XX myVar_$;
public MyClass:XX myVar()                        { return myVar_$; }
public synchronized MyClass:XX myVar_ownS$()     { return myVar_$; }
public synchronized MyClass:XX myVar_outS$()     { return myVar_$; }
public void myVar(MyClass:XX to)                 { myVar_$ = to; }
public synchronized void myVar_own(MyClass:XX to) {myVar_$ = to }
public synchronized void myVar_out(MyClass:XX to) {myVar_$ = to }
```

Separate getter/setter methods exist for the *owner* and *outsider* context if the *@NoRace* annotation is supported. If it is decided that this should never be allowed, then these can be consolidated. If *myVar* is static, then all of these methods would also be static.

If the variable is *read-only* (myVar:RR, myVar:P, or myVar:SS), then all of the setters can be eliminated. If the variable must always be synchronized (myVar:SS, myVar:SU, or myVar:US) the basic *myVar()* getter and *myVar(MyClass:TT to)* setter will also be synchronized.

This provides all necessary synchronization within the Class which defines *myVar*. At the *call site*, for a simple read the *myVar()* method will be invoked – and the caller does not need to be aware if it is synchronized. If an *S* is in effect, then either the *myVar_out$S()* or *myVar_own$S()* method is invoked (for access by an *outsider* or *owner*). Similarly, if *myVar* allows updates then either the basic *myVar(MyClass:TT to)* or *myVar_ownS$(MyClass:TT to)* or *myVar_outS$(MyClass:TT to)* method will be invoked.

If at the *call site* the compiler determines that the target variable has not been re-compiled (and flagged) as a *property*, then the *Basic Implementation* code must be generated.

Note that as in [12], the developer may also explicitly specify any of the getter and/or setter methods if desired.

The *Desired Implementation* is clearly a major, breaking change to the language and may be impossible to implement as a practical matter. The *property* concept would be a worthwhile addition in it's own right, and recover the encapsulation which was lost by making variables public and using tags to control R/W access. Execution time should be the same since most optimizing compilers will reduce the generated getter/setter methods to a simple variable fetch or assignment if possible.

Z. Target Objects

In any application designed for multi-threaded operation, most target objects referenced by a visible variable will themselves be synchronized internally. java.util.Date is an exception, unless re-compiled with the *SyncProperty* annotation. Many other standard utility objects (e.g. Lists) are also not synchronized internally, so must be synchronized if necessary or recompiled with *SyncProperty*.

AA. 64-bit Variables

Object references and most primitive values are stored atomically, and are not a concurrency issue. Visible variables declared as *long* or *double* are 64-bit values, and on some processors may be loaded/stored in 2 separate 32-bit operations. These must be declared as *volatile* (see JLS 17.7).

BB. Hiding Intermediate States

When a complex new value must be created and assigned to a (visible) variable, that variable should also be declared as *volatile* so that outsiders cannot access intermediate values during construction. For example:

```
public String[] myStrings:RW ={"1st","2nd","3rd"};
synchronized public void addString(String s){
  String[] tmp = new String[myStrings.length + 1];
(1) System.arraycopy(myStrings, 0, tmp, 0, myString.length);
(2) tmp[tmp.length - 1] = s;
(3) myStrings = tmp
}
```

If *myStrings* is declared *private* with a synchronized *getStrings* method, then this handles the concurrency. However, as listed above *myStrings* may be accessed by another thread during execution of the *addString* method and observe an intermediate state of the array.

As written, it is legal for the compiler to optimize and execute line (3) first, and then lines (1) and (2), since the result is the same in either case. However, this allows an observer in another thread to see the *myStrings* array during the construction process – it may be filled with *null* values, have some number 1..N of the original string values (as *arraycopy* is processed), or some other partial result. (See [4] for additional examples and discussion.)

The *volatile* keyword must be added to *myStrings* to ensure that all of the code before line (3) executes before line (3) makes the new value of *myStrings* visible to other threads.

CC. Modifying Multiple Variables

If a single method will modify multiple (visible) variables, the only safe way for outside code to access these variables is to synchronize on the object instance. For example,

```
public int varA:RW = …
public int varB:RW = …
synchronized public void modifyBothVariables(int valueA, int valueB) { …. }
```

Then the external code must be assured that *modifyBothVariables* is not executed by:

```
int myVarA;
int myVarB;
synchronized(targetobject){
   myVarA = targetobject.varA;
   myVarB = targetobject.varB;
}
```

Which places a burden on the caller to understand the situation. In recent versions of Java, a much better solution would be to define a *synchronized* method which returns a *Record* containing values for both *varA* and *varB*.

DD. Synchronizing java.util.Date

java.util.Date is largely deprecated, but still appears in an enormous amount of existing (and new) Java code. It is used within this document to represent a well-known Class which is designed to be mutable.

The *Date* Class itself contains no *synchronized* methods, hence instances of Date will require synchronization within any multi-threaded usage pattern or use of the *SU* tags.

IX TAGGED EXAMPLE

Using the tags, the original *Message* example can now be formulated as below - which is now perfectly safe. The Message Class exposes all of the information that any users would need, but is assured that no user can modify any value unless one of the defined mutator methods is invoked.

```
public class Message {
  /****************************************************/
  /* Message instance variables and methods           */
  /****************************************************/
  public Date:P  createdAt:P = new Date();
  public Date:SU sentAt:P    = new Date(0);
  public Date:SU ackAt:P     = new Date(0);
  public char:P[] theMessage:P;

  public Message:R(char[] msg:R)          { theMessage = Array.copyOf(msg, msg.length);
  public Message:R(char:P[] msg:R)        { theMessage = msg; }
  public boolean hasBeenSent:R()          { return sentAt.getTime() > 0; }
  public boolean hasBeenAcknowledged:R()  { return ackAt.getTime() > 0; }
  public void markSent:W(Date:R sent:R)   { sentAt.setTime(sent.getTime()); }
  public void markAck:W(Date:R acked:R)   { ackAt.setTime(acked.getTime()); }
  /****************************************************/
  /* Static variables and methods to send messages    */
  /****************************************************/
  public static int maxSentMessages:P = 24;
  public static Date:P systemStart:P = new Date();
    /** The timestamp of the last message sent */
  public static Date:SU lastSent:P = new Date(0);
    /** The most recently sent messages, up to maxSentMessages */
  public static List:SU<Message:SU> sent:P = new ArrayList<>();
    /** Messages waiting to be sent */
  public static List:SU<Message:SU> pending:P = new ArrayList<>();
   public static void sendMessage(Message msg){
      // .... logic to send the message
    }
  }
}
```

This is done in the same number of lines of code (which is *half* the lines of code required for a safe implementation without tags). The tags provide important information to any user about how variables may be modified, and which methods are mutators. (And this may be even more important to indicate the API contract to any developer maintaining this code in the future.) IDEs may also use this information to provide a more information-rich display to the user – for example, group variables and methods by mutators (W) or read only (R).

Execution time will be more efficient since in most cases a direct variable reference replaces a method call, and especially since it is not necessary to make defensive clones of instances. Synchronization overhead has been introduced since we are assuming that this Class will be used in a multi-threaded environment. If *Date* was instead a Class itself designed for multi-threading, and synchronized internally, then the *SU* tags could be *RW* instead.

Note that the *markAsSent* and *markAsAcknowledged* methods specify only an R tag for the input Date parameters since they immediately utilize the primitive information required and have no dependence on whether that value is changed in the future.

Many developers, with justification, would object that this implementation is too open and violates the concept of *information hiding* – specifically that the structure of the *sent* and *pending* lists should be encapsulated within the

Message Class. Agreed, and there is nothing in this proposal that prevents the developer from making these *private* and implementing methods to instantiate a new List to return to any caller.

That is a design decision which must trade off the increased complexity and execution time against the flexibility to make future changes to the implementation without requiring any change to the API. But also note that if initially implemented as:

```
private static List:RW<Message:RW> sent:RR = new ArrayList<Message>();
public static List:RW<Message:RW> getMessagesSent:R(){ return sent; }
```

there is no overhead needed to clone any instances. The *caller* must accept the possibility that the elements of the list may change and can make a copy if necessary. If assigned to a local variable this must be promoted to **List:RR<Message:RR>**.

If the implementation of the *sent* mechanism changes in the future, then the getMessagesSent routine can build a new List (with deep-clone Message instances) to be returned, and change the specification to:

```
public static List:P<Message:P> getMessagesSent:R()…
```

which is a non-breaking change to the API since only the *owner* tags have been changed. Some callers may have implemented a defensive copy which is now redundant.

A subtle problem occurs if the caller *depends* on the fact that new messages will be added to the list as they are sent (and updated when acknowledged). If the *sent* mechanism is changed as above, then this no longer holds. So the caller might code defensively and require an exact match:

```
private List:RR<Message:RR> mySentList:P = Message.getMessagesSent():R:[XRXW, XRXW];
```

which detects the problem at compile-time rather than as a *very* hard to find runtime bug.

X KNOWN ISSUES

There are known weaknesses with this approach. Some can be addressed by future research but others are inherent.

EE. ? Method Tag

For such a method, the developer has forced acceptance of the method (usually as read-only) even though it does not pass all required checks.

At least the ? tag alerts any caller to the situation, and the compiler generates warnings (instead of errors) pinpointing the violations. This is a major improvement over just a JavaDoc statement that a method is read-only.

FF. Native (JNI) ? Methods

The compiler can offer no assistance when dealing with native methods. Again, this situation is no worse than an existing JavaDoc claim that a method is read-only.

GG. Reflection, SecurityManager

The tags proposed here are not a guaranteed security environment. The *java.lang.reflect* methods may already be used to access a private variable, or modify a variable declared as *final*.

Additional research is needed to determine the interaction of these tags with the capabilities of the reflection mechanism, and also with any *SecurityManager* defined for an application.

XI TESTBED COMPILER

A modified Java 8 compiler is available at:

`https://drive.google.com/open?id=0B9h3YMINZ271dWcyS1RjbEZLMHc`

and contains:

- A *Start Here* text file which describes the contents of the directory, and how to execute the compiler - javac (javacTags.jar) - and javap utility (javapTags.jar)

- A Report Card which provides details of the features supported (fully, partially, or not).
- A Compiler Changes document which discusses the changes made to the compiler, the compiled .class file, and the javap utility.
- The modified source code for the compiler and javap.
- A series of test applications.

See the Appendix for more details.

XII TRANSITION

Any adoption of the tags proposed here will be difficult since the transition must proceed in a bottom-up fashion. For example, if writing a new tagged application with a read-only method which calls the *size()* method on a standard java.util.LinkedList, the compile will fail since *size()* will not be tagged as :R.

Also, some libraries which are re-factored to fully incorporate tags will migrate some variables which were previously *private* – with a *getter* and *setter* – and these will become *public* with :RW tags and drop the *getter* method.

The transition might be supported by some utilities and compile-time options and annotations.

HH. The JavaTag Utility

A standalone utility which will process compiled *Classes* or *libraries,* analyze the compiled code, and produce output Classes/libraries with any tags which can be deduced. It will determine the most restrictive possible tags for a source which does not specify **any** tags, and should generate an error if some tags are present.

This will allow existing libraries to be reprocessed to produce *tagged* output Classes. However, wherever possible the source should be updated because the tags are important documentation for the users and maintainers of the library.

The most important result will be the detection (and tagging) of *read-only methods*. In existing code, most variables will be *private* with a *getter* method, and these will be identified as read-only. (And may well produce some surprises!) Only an R tag will ever be inferred for a method. Inferring a T (transient) tag may produce erroneous results. For example, in the java.util.Date Class all variables are listed as *transient* so the *setTime(….)* method would be inferred to be immutable (T) if allowed.

Processing all of the standard Java libraries, and many popular 3rd party libraries (e.g. apache.commons) through this utility should provide a basis to support the development of tagged applications. For example, the *size()* method of LinkedList is expected to be tagged as read-only (and lead to some astonishment if it is not!).

II. Compilation Options, Annotations

The compiler will verify any specifications present in the source and generate warnings and errors as needed.

Several options are envisioned for the compiler – especially during any transition when some source code has been *tagged* but must interface to *untagged* modules or libraries. In some cases, these may be specified as either a command line option or an annotation in the source – for example:

- *ignoreTags* – ignore all tags in the source and compile as before (or assume that all tags are WW).
- *inferTags* – implement the logic from the *JavaTag* utility dynamically at compile time (most likely when loading a Class from an untagged library).
- *tags(class.<name>:xx:xx)* – infer that the named variable (or method) should be compiled as-if the given tags were present.

XIII FUTURE RESEARCH

JJ. Migrate to recent JDK

The proof of concept implementation is based on Java 8. This needs to be migrated at least to JDK 11 (the current LTS release), JDK 14 (current release), or JDK 17 (the next LTS release, planned for September 2021).

KK. Implement a Significant Application

Once migrated to a recent version of the JDK, use tags for the implementation of a non-trivial project and report on the experience. This might well be the compiler itself.

LL. Change the Default

The current *WW* default enables the correct compilation of existing source which is un-tagged. Once most source has been upgraded to use R/W tags, changing the default (either intrinsically or by compiler options) could be:

- Default to *RR* – so that only *W* (mutable) tags need to be specified.
- Default to *P* – which essentially has Java operate as a *Functional Programming* language.

MM. Referentially Transparent Methods

This would define a tag to specify that if a given method is called with the same parameter values it will always return the same value. (Most mathematical functions – e.g. Math.sqrt(x) – are referentially transparent.) (There are numerous discussions in the Functional Programming literature [1].)

This can be a major performance improvement since the result of an expensive operation can be cached once computed the first time – e.g. compute the value of Pi to 999 decimal digits. It may also allow the compiler to perform additional optimizations – e.g. *Math.sqrt(x*) inside a loop where 'x' is invariant may be moved outside the loop if known to be referentially transparent.

It is expected that this tag would be a declaration by the developer since it may not be possible for the compiler to verify referential integrity for an arbitrary method.

NN. Uniform Access Principle, Property Logic

A longstanding shortcoming of Java is that if during the refactoring of a Class it is necessary to change a variable from public access to private, and add a getter and setter to manage access (or vice-versa), this is a breaking change to the API of the Class.

This will become more of a problem using *tags* since many variables which would previously be defined as private with a getter/setter, will now be declared as public with an RW tag. Any future refactoring then has an increased chance of a breaking change.

Merging the *uniform access principle* introduced in [11] and the *property* concept within Java would be an elegant solution to this issue, and a worthwhile addition independently, but is beyond the scope of this discussion.

OO. Enhanced ClassLoader

Determine if it is possible to enhance the ClassLoader to detect any RHS tag violation during the loading of a Class. Even if possible, this might prove to be prohibitively expensive, and might be controlled by a JVM setting.

PP. Verify Transient

Verify if a method which may modify *transient* variables can safely allow a *Pure* determination – primarily considering cache coherence issues in a multi-processor context.

XIV CONCLUSION

This proposal merges the concept of immutability into the most widely used object-oriented language. Overall, these changes are extensive, and implementation of all particulars may not be possible in the real world – especially since some may represent breaking changes to existing code. Summarizing some of the major topics – in order of importance:

QQ. Tags on Types

This (especially the basic R/W tags), in the view of the author, is the most important contribution of this paper. It allows a Class or method to expose a ***reference*** to an instance of a Class, but require read-only access to that instance (through that reference). This eliminates the requirement that instances be cloned in order to provide safe access, which improves performance, reduces code complexity, and makes this critical constraint visible to any user.

Returning *Date:RW* from a method also implies that the owner of the instance may modify the value in the future, *Date:RR* implies that there will not be any modification by the owner, and *Date:WR* implies that the owner is passing control of the target instance to the caller. Note, however, that these implications are not guaranteed by the compiler (although returning *Date:P* does include a guarantee).

This is especially important since the concept of *cumulative immutability* allows a single reference to impose immutable access to a large structure containing thousands of instances – for example the *HTMLNode:RR rootNode* discussed previously.

It also allows forcing immutablility on an *instance* of a Class which is normally mutable – without requiring redundant implementations of the Class or the use of wrappers (which will usually throw exceptions at runtime instead of raising an error during compilation).

RR. Tags on Methods

Tagging a method as read-only (R) is necessary in order to support RR tags on types, allows the compiler to validate this important requirement, and provides important information to any developer invoking that method. The *?* tag is necessary in many instances to force treatment of a method as immutable (especially for JNI methods).

SS. Transfer of Ownership

This is intended to support a couple of important edge cases, but is not intrinsic to the most critical features of this proposal. It is intended to cleanly support the use of *factory methods* and the common situation where a caller instantiates an instance but then intends to immediately pass control of this instance to another Class. For example, creating a *Message* instance so that it can be passed to the *Message.sendMessage(…)* method.

TT. Tags on Variables

These are provided for completeness, to allow for the reduction of boilerplate code, and to provide a more descriptive API for any user (or maintainer) of a Class.

However, these could be eliminated, and rely on the current mechanism of private variables and getters/setters used to control access, without damage to the primary benefit of *Tags on Types*.

UU. Right Hand Side Tags

These are not strictly necessary, but do provide a way to detect that the implied *contract* of an API has been broken. For example, invoking a heavily used method which was previously read-only but has been changed and is now mutating, or is now synchronized, may have a drastic performance implication in a heavily concurrent application. RHS tags detect the problem during compilation instead of production.

VV. Class/Interface Tags

Also not strictly necessary, but provide any reader an overview of the structure of the Class/Interface and allow the compiler to detect implementation errors (especially during future maintenance cycles). They are most useful to declare that static methods and/or instances of a Class are designed to be *immutable* – [P, *] [P, *].

WW. Pure Instances

This is the most problematic suggestion, especially the JVM changes which would be required to support *Pure at Runtime* and changes to *java.lang.Object.*

The benefits would be primarily runtime improvements derived from reduced synchronization/coherence overhead, but require additional research and would probably vary widely between different applications.

APPENDIX

A Java 8 compiler was modified as a Proof of Concept for this proposal. This also involved adding information to the compiled .**class** file, and re-parsing this information when the Class is re-loaded by the compiler – for example, tagged Class A invokes tagged Class B. The *javap* utility was also modified in order to display the modified .class files.

XX. General Notes

- These modifications to the Java 1.8 compiler should be considered fragile. In particular, the level of testing is marginal.
- See the companion *Compiler Changes 0.1* document for details about how the compiler was modified to support the R/W tags, and the tests applied.
- Note, as discussed in *Compiler Changes*, that most R/W tag logic is enforced at compile time and the output ***.class** files should execute on the existing JVM. The *Pure at Runtime* concepts will require JVM enhancements.

YY. Features Supported

The features supported by this version of the compiler are listed generally in the order in which they are discussed in this document. An overall letter grade is provided along with further discussion as needed:

A	Fully supported.
B	Partially supported. All appearances should be parsed correctly, but not all semantic rules are applied.
C	Parsed and ignored. Tags in the source should be parsed without error, but no semantic logic is implemented.
D	Unsupported. Tags in the source will usually result in a compile-time error.

The following table is the *Report Card* of the support of these proposed features as of this document. A current version is present on the shared drive with the compiler.

Topic	Grade	Comments
Variables	A	
Variable Assignment Rules	A	
Type Specifications	A	
	D	S and U tags
	D	Exact R/W matches on RHS not supported (e.g. RXWX)
Pure Instances	B	*Pure at Birth* supported. *Pure at Runtime* not supported.
Arrays	A	
Array Assignments	A	
Generics	A	Generic declarations & reflect tags in .class file
	D	Does **not** accept tags (e.g. <T:R>) and either verify (or supply) tags at each instance of T in the code. All entries of T must be hand-coded with the :R tags.
	C	Does not determine *R-Safe* or generate a compile time error if a generic which is not R-Safe is instantiated with a type which specifies an :R or :P tag.
	B	Verify R/W tags on parameters of calls & returns when a generic type is involved
Generic Methods	C	
Type Assignment Rules	A	
Downcasts and Upcasts	A	
Methods	A	Verification of :R methods as read only (:F for now == R?)
	C	Transfer of ownership not yet supported
	D	Other tags beyond W, R, and F (will become R?)
Method Parameters	B	Varargs not yet supported
Method Overloading	D	
Method Overriding	B	Varargs not yet supported
Return Types	A	
	C	Transfer of ownership
Scope Implications	D	
Lambdas	C	
Constructors	B	Does not support *return tags* after the method tag.
Class and Interface Tags	D	
Pure at Runtime	D	Requires JVM support
Concurrency Issues	D	Potential optimizations once R/W tags are fully supported.
Compilation Options	C	-ignoreTags compiler option supported.
Annotations	D	
JavaTag Utility	D	

ACKNOWLEDGEMENT

Dr Douglas Lyon at Fairfield University and John P Crowley at Bank of America reviewed early versions of this manuscript and provided important suggestions.

RELATED WORK

Almost all of the *Functional Programming* languages embody, and are built on, the concepts of immutable objects and read-only methods. The concepts here are an attempt to import many of these into Java as declarative enhancements to the language. An immense body of work exists around this topic – instead of repeating this here, some introductory references are included which in turn act as pointers to the main bodies of work [1].

Both the C and C++ languages include a *const* keyword to specify that references or methods are immutable [2]. Neither of these include the concept of *outsiders* versus *owners* as distinct environments, which limits the use of *const*. (See the Related Work section in [3] for a more thorough explanation.)

ConstJava is a comprehensive attempt to apply the *const* keyword concepts to Java [3]. This paper also has an extensive bibliography, and some experimental results from applying these concepts. [These same experiments need to be carried out for the concepts presented here.]

[4] is a discussion of the shortcomings of several earlier attempts to add *readonly* to Java and also includes an extensive bibliography.

Uniqueness and Reference Immutability for Safe Parallism is defined in[7], and inspired the concept of *Pure at Runtime*.

REFERENCES

[JLS] The Java Language Specification (SE8) – https://docs.oracle.com/javase/specs/

[1] https://en.wikipedia.org/ wiki/Functional_programming

[2] The C++ Programming Language, special edition, 2000. Bjarne Stroustrup

[3] http://types.cs.washington.edu/ javari/constjava/ConstJava.pdf)

[4] http://www.jot.fm/issues/issue_2006_06/article1/

[5] https://en.wikipedia.org/wiki/Java_Modeling_Language

[6] Programming in Scala, 2nd Edition, Odersky, Spoon, Venners

[7] https://www.microsoft.com/en-us/research/publication/uniqueness-and-reference-immutability-for-safe-parallelism/

[8] https://docs.oracle.com/ javase/tutorial/java/generics/index.html

[9] http://www.angelikalanger.com/ GenericsFAQ/FAQSections/TechnicalDetails.html

[10] https://www.rust-lang.org/ enUS/documentation.html)

[11] Object-Oriented Software Construction, Meyer

[12] C# 6.0 in a Nutshell, J Albahari and B Albahari

[13] Delphi 2 Developer's Guide, 2nd Edition, Pacheco and Teixeira

John D Crowley BS Physics/Math, Boston College, MS and PhD Computer Science, University of Pennsylvania.
40+ years of application architecture and development, with over 2M lines of code in multiple languages. Multiple startups.