# The Deterministic Parallel Java Language Reference Manual Version 1.1

Revised January 2013

## Contents

# 1 Introduction

This document is a reference guide to the Deterministic Parallel Java programming language, Version 1.1 (DPJ v1.1). It explains in detail the new language features that DPJ adds to Java, and how they work. It is intended to be accessible to readers with no previous knowledge of DPJ, though familiarity with Java is assumed.

Many cross-references appear throughout the document. To refer to sections, we use the section symbol §: for example, § 5.1 refers to the section numbered 5.1.

Readers of this document may also wish to consult the following:

- *The Deterministic Parallel Java Tutorial* provides a tutorial introduction to DPJ, and explains how to write some common parallel patterns in DPJ. The DPJ programmer should probably read that document first. It cross-references this document, so you can look up particular features that you want to learn more about.
- *The Deterministic Parallel Java Installation Manual* explains how to get started installing the `dpjc` compiler, using the compiler to compile DPJ programs, and running DPJ programs.

The rest of the document is organized as follows:

- § 2, *Region Path Lists*, describes region path lists, or RPLs. These are the general form of a region name in DPJ. Regions partition the heap and help express heap effects. RPLs are hierarchically structured, and the structure allows many fine distinctions between sets of regions to be expressed.
- § 3, *Classes and Interfaces*, describes DPJ's extensions to Java classes and interfaces. These include features for declaring region names, assigning region names to class fields, summarizing method effects, and writing classes and methods with region variables.
- § 4, *Types*, describes DPJ's extensions to Java's class and array types. The main difference is that both class and array types can have region arguments. There are also new rules for type comparisons, casts, etc. to support the extensions.
- § 5, *Effects*, describes DPJ's effect system, which is closely integrated with the regions in the type system. Effects describe operations on the heap. The programmer declares the effects of methods, and the compiler infers the rest of the effects. The compiler checks to make sure that the effects of parallel tasks are mutually noninterfering.
- § 6, *Parallel Control Flow*, describes DPJ's constructs for creating parallel tasks.
- § 7, *The DPJ Runtime*, explains the classes in the DPJ runtime that are useful for manipulating arrays.
- § 8, *Exception Behavior*, explains what happens when an exception is thrown in DPJ.

# 2 Region Path Lists (RPLs)

This section describes *region path lists*, or RPLs. RPLs are the fundamental tool for describing sets of memory locations in DPJ. RPLs can be *fully specified* or *partially specified*. A fully-specified RPL names a *region*, which represents a set of memory locations on the heap. The correspondence between memory locations and regions is given by the way that the programmer has assigned RPLs to class fields (§ 3.2) and bound RPLs to region parameters in class types (§ 4.1.1) and method invocations (§ 3.4.2). A partially-specified RPL names a *set of regions* and is useful for expressing hierarchical effects (e.g., "all regions under this one").

RPLs are naturally nested. In fact the motivation for name "region path list" is that an RPL is a sequence of names that describes a path from the root in a tree of regions. Together with partially specified RPLs, this nesting structure provides a great deal of power in comparing sets of memory locations to determine whether they are overlapping or disjoint. These properties are the key to specifying and checking effects. In particular, checking *subeffects* (e.g., that a method's effect summary covers its effects, § 3.3.1) requires reasoning about inclusion of RPLs, while checking *noninterfering effects* (e.g., that the branches of a `cobegin` statement do not interfere, § 6.1) requires reasoning about disjointness of RPLs.

The rest of this section proceeds as follows. § 2.1 describes the simplest form of RPL, which is just a name, such as a region name declared at class scope (§ 3.1). § 2.2 describes how to construct RPLs using *sequences of names*, which put a nesting structure on RPLs. § 2.3 describes the use of region parameters (§ 3.4) with RPLs. These parameters allow RPLs to vary with the class type or method invocation. § 2.4 explains how to write *partially specified RPLs*, which describe sets of fully specified RPLs. § 2.6 explains the rules for comparing RPLs for properties such as inclusion and disjointness.

Finally, a word about the uses of RPLs. This section is about the *definition* of RPLs, and the relations among them (inclusion, disjointness, etc.). The various ways in which RPLs can be *used* (i.e., how they actually appear in a DPJ program) are covered in other sections. To provide context for this section, we list all the possible uses here, together with the sections discussing those uses:

- RPLs can appear in field RPL specifiers, to assign class fields to regions (§ 3.2).
- RPLs can appear in effect summaries (§ 5.2). These RPLs describe the effects of a code statement.
- RPLs can appear as arguments to parametric class types (§ 4.1.1). These RPLs provide actual regions for the parameters in the instantiation of a class or array.
- RPLs can appear in method invocations, as arguments to method region parameters (§ 3.4.2). These RPLs provide actual regions for the parameters in an invocation of a method.

Throughout this section, we provide example uses to illustrate the definitions in context.

## 2.1 Basic Region Names

This section describes the simplest form of RPL, which is a single name. There are five kinds of region names:

1. `Root`, a special name that is always in scope.
2. A region name declared at class scope (§ 3.1).
3. An array index region `[`*exp*`]`, where *exp* is an integer expression.
4. A region name declared in a method-local scope (§ 3.3.3).
5. A `final` local variable or `this`.

The first three kinds of names are available everywhere (except that class region names may have access qualifiers, see § 3.1). The last two kinds of names are available only in the scope where the region names or variables are active.

### 2.1.1 The Name `Root`

The region name `Root` is always in scope, and can be used anywhere as an RPL. For example, the following code uses `Root` in a field region specifier (§ 3.2) and to instantiate a class type (§ 4.1.1):

```
class RootExample<region R> {
    RootExample<Root> x in Root;
}
```

The name `Root` has a special meaning: it is the root of the region tree. Every RPL is nested under `Root`. For more information about the nesting relation on RPLs, see § 2.6.2.

### 2.1.2 Class Region Names

A region name declared as a class member (§ 3.1) is available for use as an RPL in any scope where it is visible. For example, the following code declares a region name `ClassRegion` at class scope, then uses it in a field region specifier (§ 3.2) and to instantiate a class type (§ 4.1.1):

```
class ClassRegionExample<region R> {
    region ClassRegion;
    int ClassRegionExample<ClassRegion> x in ClassRegion;
}
```

Because class region names can be package- and class-qualified, like Java static fields, the following code is also legal:

```
class Class1 {
    region ClassRegion;
}
class Class2 {
    int x in Class1.ClassRegion;
}
```

When a class region name $R$ is used as an RPL, it is shorthand for the name sequence `Root:`$R$, as described in § 2.2. For example, the following definition of `Class2` is equivalent to the one given above:

```
class Class2 {
    int x in Root:Class1.ClassRegion;
}
```

### 2.1.3 Array Index RPLs

The name `[`*e*`]`, where *e* is an integer expression, functions as an RPL, called an *array index RPL*. The array index RPL represents a region indexed by the number that *e* evaluates to at runtime (so there is a different region for each natural number). It is useful in conjunction with *index-parameterized arrays* (§ 3.5.3) and with DPJ's built-in features for array partitioning (§ 7.2).

As with class region names used as RPLs (§ 2.1.2), an array index RPL `[`*e*`]` is short for `Root:[`*e*`]`.

### 2.1.4 Local Region Names

A region name declared inside a method body (§ 3.3.3) is available for use as an RPL in the scope where it is visible. For example, the following code declares a region name `LocalRegion` in a local scope, and uses it to instantiate a type:

```
class LocalRegionExample<region R> {
    void method() {
        region LocalRegion;
        LocalRegionExample<LocalRegion> lre =
          new LocalRegionExample<LocalRegion>();
    }
}
```

As with class region names and array index regions, an RPL consisting of a single local region name implicitly starts with `Root`. For example, the class definition above is equivalent to the following:

```
class LocalRegionExample<region R> {
    void method() {
        region LocalRegion;
        LocalRegionExample<Root:LocalRegion> lre =
          new LocalRegionExample<Root:LocalRegion>();
    }
}
```

### 2.1.5 `final` Local Variables and `this` as Region Names

The following variables may be used as an RPL:

- A `final` local variable or method parameter with a class type.
- The variable `this`, in a non-static context.

This usage is called a *variable RPL*. For example, the following code uses `this` as an RPL in a field region specifier (§ 3.2), as an argument to a class type (§ 4.1.1), and in an effect summary (§ 5.2):

```
class VarRegionExample<region R> {
    VarRegionExample<this> x in this;
    void setX(VarRegionExample<this> x) writes this {
        this.x = x;
    }
}
```

Variable RPLs allow *runtime objects to function as regions*, as in type systems based on *ownership*. The variable stands in for the object stored to the variable at runtime. Because the variable must be `final` or `this` (which cannot be assigned to), the variable always refers to the same object. This feature is particularly useful in conjunction with the `DPJParititon` class (§ 7.2), because it provides a way to instantiate a partition with the region corresponding to the array being partitioned.

As in ownership systems, a variable RPL is nested under the first RPL argument of its type, which is called the *owner RPL* of the type (§ 4.1.4). For example, in the code above, the variable `this` has type `VarRegionExample<R>`, so it is nested under the RPL `R`. For more information about the nesting relationship on RPLs, see § 2.6.2.

## 2.2 Sequences of Basic Names

RPLs are built up from colon-separated sequences of names; this gives them a natural nesting structure. For example, if `A` and `B` are names, then `A`, `B`, `A:B`, `B:A`, `A:A:B`, and so forth are all RPLs. The nesting structure is given by the syntax: for example, `A:B` and `A:A:B` are both nested under `A`. Note that nesting does not imply inclusion; `A` and `A:B` are distinct regions. In particular the effect `writes A` does *not* cover (or imply) the effect `writes A:B`. However, the nesting structure creates a hierarchy of regions that is useful in conjunction with partially specified RPLs (§ 2.4), because then we can use it to say things like "all RPLs under this one" or "all RPLs under this one that end in `B`." Separating nesting from inclusion like this makes RPLs more complicated, but it also allows greater precision.

An RPL is valid if it is conforms to the following rules:

- It is composed of basic names (§ 2.1), separated by colons.
- If `Root` (§ 2.1.1) or a variable name (§ 2.1.5) appears in the sequence, it must appear first (so, in particular, both cannot appear).

The names in the sequence are called the *elements* of the RPL. If an RPL does not start with `Root` or a variable element, then it is treated as implicitly starting with `Root`. For example, `A` is the same as `Root:A`, and `[0]` is the same as `Root:[0]`.

The following are examples of valid RPLs, assuming that `A` and `B` are declared region names in scope:

```
A         // Declared name
Root:A    // Same as A
A:B       // Sequence of names
Root:A:B  // Same as A:B
[0]       // Array index region
Root:[0]  // Same as [0]
this      // Variable region
this:A    // Variable region with A appended
```

The following are examples of RPLs that are *not* valid, because they do not conform to the rules above:

```
A:Root     // Root must appear first
this:Root  // Root must appear first
A:this     // this must appear first
Root:this  // this must appear first
```

## 2.3 Parameterized RPLs

Class and method region parameters (§ 3.4) work with RPLs in a natural way. First, a region parameter standing alone is always a valid RPL, as shown in the examples in §§ 3.4.1 and 3.4.2. Second, an RPL may be constructed from a region parameter followed by a colon-separated sequence of elements, similar to a sequence of basic names (§ 2.2). If a parameter appears in an RPL, then it must appear first, and no `Root` or `final` local variable may appear in the RPL (because those, when they appear, must also be first).

The following are examples of valid parameterized RPLs, assuming that `R` is a region parameter in scope and `A` and `B` are declared region names in scope:

```
R       // Region parameter alone
R:A     // Region parameter with name appended
R:A:B   // Region parameter with names appended
R:[0]   // Region parameter with index appended
```

The following are examples of RPLs that are *not* valid, because they do not conform to the rules above:

```
A:R       // R must appear first
Root:R    // R must appear first
R:Root    // Root must appear first
R:this    // this must appear first
```

The meaning of a parameterized RPL is that at runtime, the parameter will be substituted away, generating a region that has `Root` or an object as its first element. For example, consider the following class definition:

```
class ParameterExample<region R> {
    region r;
    int x in R:r
}
```

The class `ParameterExample` has one region parameter `R`, and one field `x` placed in RPL `R` using a field RPL specifier (§ 3.1). Suppose we now instantiate that class to a type (§ 4.1.1), by binding `Root` to `R`:

```
ParameterExample<Root> pa = new ParameterExample<Root>();
```

This type says that the object bound to `pa` at runtime has its field `x` in RPL `Root:r` (substituting `Root` for `R` in `R:r`). For more information about how this subsitution works, see § 4.3.

## 2.4 Partially Specified RPLs

In order for the nested structure of RPLs to be useful, the programmer must be able to name a *set* of RPLs such as "all RPLs nested under this one." To support this kind of naming, DPJ has a special element `*` that stands in for any sequence of RPL elements. There is also a wildcard array index element `[?]`, where the `?` stands in for any natural number. This is useful for naming sets of array regions.

### 2.4.1 The * RPL Element

An RPL may be written with `*` as one if its elements. The RPL must otherwise follow the rules in §§ 2.2 and 2.3: for example, if a parameter appears in the RPL, it must be first. It is permissible for an RPL to start with `*`; in this case, there is an implicit `Root` before the `*`. In particular, the RPL `*` is equivalent to `Root:*` and refers to all regions.

The meaning of an RPL containing `*` is that it stands in for all legal RPLs that could be constructed by substituting sequences of zero or more elements for the `*`. For example, `A:*:B` stands in for `A:B`, `A:A:B`, `A:B:B`, `A:A:A:B`, etc. However, `A:*:B` does *not* stand in for `A:Root:B`, because that is not a valid RPL (`Root` must appear first). See §§ 2.6.2 and 2.6.3 for more information about the nesting and inclusion rules for RPLs using `*`.

Here are some examples of valid RPLs constructed with the `*` element, assuming that `A` and `B` are declared region names in scope:

```
*           // All RPLs
Root:*      // Same as *
*:A         // All RPLs under Root ending in A
A:*         // All RPLs under Root:A
Root:A:*    // Same as A:*
this:*      // All RPLs under this
this:*:A    // All RPLs under this ending in A
```

Although more complex RPLs are possible, typically in user code at most one * appears in an RPL, as shown in the examples above.

Here are some examples of RPLs that are *not* valid, because they do not conform to the rules stated in §§ 2.2 and 2.3 (`R` is a region parameter):

```
*:R         // R must appear first
*:this      // this must appear first
A:*:Root    // Root must appear first
```

### 2.4.2 The `[?]` RPL Element

The *wildcard array index element* `[?]` may be used in an RPL anywhere that an array index element `[e]` may appear (§ 2.1.3). It stands in for any array index element. For example:

```
[?]         // Matches [e] for any [e]
Root:[?]    // Same as [?]
[?]:A       // Matches [e]:A for any e
this:[?]    // Matches this:[e] for any e
```

## 2.5 Local RPLs

A *local RPL* is an RPL that contains a local region name (§ 2.1.4) as an element. It may be parameterized and/or partially specified. For example:

```
class LocalRPLs<region R> {
    region ClassRegion;
    void method() {
        region LocalRegion;
        // LocalRegion is a local RPL; it is equivalent
        // to Root:LocalRegion
        LocalRPLs<LocalRegion> x = null;
        // R:LocalRegion:* is a local RPL
        LocalRpls<ClassRegion:LocalRegion:*> y = null;
        // ClassRegion is not a local RPL
        LocalRpls<ClassRegion> z = null;
    }
}
```

Local RPLs are important because they define *local effects* (§ 5.3). A local effect is visible only in a method and its callees, so doesn't have to be reported to the caller. For example, in the code above, any effects on `LocalRegion` or `R:LocalRegion` could be omitted from the effect summary of `method`.

## 2.6 Comparing RPLs

To reason about effects, both the programmer and the compiler have to make judgments like "this RPL is included in that one" or "this RPL is disjoint from that one" (where "disjoint" means that the corresponding sets of memory locations are nonintersecting). The programmer has to do this in order to reason about which

code sections may be run in parallel safely, i.e., without interference. And the compiler has to do it to check that the programmer got it right.

To support this kind of reasoning, there are four relations on RPLs:

1. *Equivalence*, meaning that two RPLs refer to the same set of regions.
2. *Nesting*, meaning that one RPL is a descendant of another in the region tree. For example, `A:B` is nested under `A`.
3. *Inclusion*, meaning that the set of memory locations represented by one RPL includes the set of locations represented by another. For example, `A:B` is included in `A:*`. Note that *`A:B` is not included in `A`*. The `*` is necessary to get the inclusion. This is to increase the precision of effect specifications with RPLs.
4. *Disjointness*, meaning that the set of memory locations represented by one RPL has no common elements with the set of memory locations represented by another RPL. For example, if `A` and `B` are distinct names, then they represent disjoint regions. Or `R1` and `R2` are disjoint if they are parameters declared to be disjoint (§ 3.4.3).

### 2.6.1 Equivalence

**Equivalent RPLs:** The meaning of the equivalence relation on RPLs is that the sets of regions represented by the RPLs are the same. Two RPLs are equivalent if they have the same number of elements, and the corresponding pairs of elements are equivalent, as defined below. This test is done after adding the implicit `Root` to the front of RPLs that begin with a declared name element, array index element, `*`, or `[?]`.

For example, the following pairs of RPLs are equivalent:

- `Root:A` and `A`
- `this:[i]` and `this:[i]`, if `i` is a `final` local variable or method parameter
- `R:[i+1]` and `R:[i+1]`, if `i` is a `final` local variable or method parameter

The following pairs of RPLs are not equivalent:

- `Root:A` and `Root:A:B`
- `this:[i]` and `this:[i]`, if `i` is not a `final` local variable or method parameter
- `this:[i]` and `this:[j]`
- `R:[i+1]` and `R:[i+1]`, if `i` is not a `final` local variable or method parameter
- `R:[i+1]` and `R:[i+2]`

**Equivalent RPL elements:** The following rules govern equivalence of RPL elements.
*Basic names other than array index elements:* Two RPL elements are equivalent if they refer to the same basic name (§ 2.1), except for array index elements.
**The elements `*` and `[?]`:** Two RPL elements are equivalent if they are both `*` or `[?]`.
*Array index elements:* Two array index elements `[`$e_1$`]` and `[`$e_2$`]` equivalent if $e_1$ and $e_2$ always evaluate to the same value (i.e., are "always equal").

Since the compiler can't always know when two expressions evaluate to the same value, it applies the following conservative rules to identify pairs of always-equal expressions:

- $e_1$ and $e_2$ are always-equal if they refer to the same compile-time integer constant.
- $e_1$ and $e_2$ are always-equal if they are the same local variable or method parameter, and the variable or parameter is declared `final`.
- $e_1$ and $e_2$ are always-equal if they represent the same binary operation, and the corresponding subexpressions are always-equal. For example, `i+1` and `i+1` are always-equal, if `i` is a `final` local variable.

Otherwise, the compiler conservatively assumes that the expressions are not always-equal.

### 2.6.2 Nesting

The meaning of the nesting relation on RPLs is that one RPL is a descendant of another in the region tree. Nesting is important because, in conjunction with partially specified RPLs, it defines inclusion of RPLs (§ 2.6.3). Four rules govern nesting of RPLs.
**Nesting under `Root`:** Any RPL is nested under `Root`.
**Nesting by syntax:** One RPL is nested under another if the second is a prefix of the first. For example, `A:B` is nested under `A`, `R:A` is nested under `R`, and `this:[0]` is nested under `this`.
**Nesting by inclusion:** If one RPL is included in another, then the first RPL is nested under the second. For example, `A:B` is included in `A:*` (see § 2.6.3), so `A:B` is nested under `A:*`.
**Nesting by ownership:** An RPL that starts with a variable element (§ 2.1.5) is nested under the owner RPL of the variable (§ 4.1.4). For example, in the following code, the method parameter `param` has type `NestByOwnership<R>`, so it is nested under `R` when used as an RPL:

```
class NestByOwnership<region R> {
    <region R>method(final NestByOwnership<R> param) {
        // param is used as an RPL here; it is nested under R
        NestByOwnership<param> localVar = new NestByOwnership<param>();
    }
}
```

### 2.6.3 Inclusion

The meaning of the inclusion relation on RPLs is that the set of regions represented by one RPL is included (in the sense of set inclusion) in the set of regions represented by another RPL. Inclusion of RPLs is important in checking type comparisons (§§ 4.1.2, 4.2.2) and subeffects (§ 5.6). The following rules govern inclusion of RPLs.
**Inclusion by equivalence:** One RPL is included in another if the RPLs are equivalent (§ 2.6.1).
**Inclusion by trailing `*`:** One RPL is included in another if the second ends with `*`, and the first is nested under the second after the trailing `*` is removed. For example, `A:B:C` is included in `A:*`, because `A:B:C` is nested under `A` (i.e., `A:*` with the `*` removed).
**Inclusion by trailing elements other than `*`:** One RPL is included in another if the last element of the first is included in the last element of the second, and inclusion holds for the RPLs after stripping the last elements of both. One RPL element is included in another if the elements are equivalent (§ 2.6.1) or the first element is any array index element or `[?]`, and the second is `[?]`. For example:

- `A:B` is included in `A:*:B`, because `B` is included in `B`, and `A` is included in `A:*`
- `A:[0]` is included in `A:*:[?]`, because `[0]` is included in `[?]`, and `A` is included in `A:*`.
- `A:*:B` is not included in `A:B`, because `A:*` is not included in `A`.
- `A:[?]` is not included in `A:[0]`, because `[?]` is not included in `[0]`.

**Inclusion by constrant:** Two RPLs may be constrained to be included, one in the other. In the current language specification, the only way this can happen is when capturing a type (§ 4.3). There is no way for the programmer to directly write an inclusion constraint on RPLs.

### 2.6.4 Disjointness

The meaning of the disjointness relation on RPLs is that the sets of regions represented by the RPLs have no common elements. This relation is important in checking noninterference (§ 5.7).
**Disjoint RPLs:** The following rules govern disjointness of RPLs.
*Distinctions from the left:* Two RPLs are disjoint if they start with a sequence of one or more equivalent elements, and then their elements are disjoint, where disjoint elements are defined below. An application of

this rule is called a "distinction from the left," because it uses a difference in the RPLs starting on the left to infer disjointness. This rule works because RPLs form a tree, so two RPLs that start the same then diverge must be on different branches of the tree.

The following are examples of distinctions from the left, assuming `A` and `B` are class region names (§ 2.1.2):

- `A` and `B` are disjoint, because they are short for `Root:A` and `Root:B`, so they start with the same element `Root` and then diverge. `A:*` and `B:*` are also disjoint, for the same reason.
- If `R` is a region parameter, then `R:A` and `R:B` are disjoint, as are `R:A:*` and `R:B:*`.
- `this:A` and `this:B` are disjoint, as are `this:A:*` and `this:B:*`.

Note that there must be at least one equivalent element for a distinction to from the left. For example, region parameters `R1` and `R2` are not disjoint unless they are constrained to be disjoint (see below).

*Distinctions from the right:* Two RPLs are disjoint if they end with disjoint elements, where disjoint elements are defined below. An application of this rule is called a "distinction from the right," because it uses a difference in the RPLs starting on the right to infer disjointness. This rule works because two RPLs that end in disjoint elements can never refer to the same regions, even after substituting for parameters, `*`, and `[?]`.

The following are examples of distinctions from the right, assuming `A` and `B` are class region names (§ 2.1.2):

- `Root:*:A` and `Root:*:B` are disjoint RPLs.
- `R:*:A` and `R:*:B` are disjoint RPLs.
- `this:*:A` and `this:*:B` are disjoint RPLs.

*Disjointness by constraint:* Two RPLs are disjoint if they are each included in another RPL (2.6.3) and the including RPLs are constrained to be disjoint (3.4.3). For example, in the following code, `R1:r` and `R2:r` are disjoint RPLs, because `R1:r` is included in `R1:*`, `R2:r` is included in `R2:*`, and `R1:*` and `R2:*` are constrained to be disjoint.

```
class DisjointConstraints<region R1, R2 | R1:* # R2:*> {
    region r;
    void method() {
        DisjointConstraints<R1:r> dc1 =
            new DisjointConstraints<R1:r>();
        DisjointConstraints<R2:r> dc2 =
            new DisjointConstraints<R2:r>();
    }
}
```

Note that `R1:r` and `R2:r` would *not* be disjoint if the constraint were the weaker `R1 # R2`; the `*` is required. This is because nesting does not imply inclusion. For example, setting `R1` = `A` and `R2` = `A:B` satisfies `R1 # R2`. But then `R1:B` = `A:B` is not disjoint from `R2`, after substituting `A` for `R1` and `A:B` for `R2`. However, `R1` = `A` and `R2` = `A:B` does not satisfy `R1:* # R2:*`, because `A:B` is included in `A:*`.

**Disjoint RPL elements:** Two RPL elements are disjoint if one is a name and the other is an array index (including `[?]`); or they are different names; or they are the array index elements $[e_1]$ and $[e_2]$, where $e_1$ and $e_2$ always evaluate to different values at runtime (i.e., are "never-equal expressions").

Since the compiler can't always know when two expressions evaluate to the same value, it applies the following conservative rules to identify pairs of never-equal expressions:

- $e_1$ and $e_2$ are never-equal if they refer to different compile-time constants. For example, 0 and 1 are never-equal.
- $e_1$ and $e_2$ are never-equal if they represent the same binary operation, one of the corresponding subexpressions is always-equal (§ 2.6.1), and the other is never-equal. For example, if `i` is a `final` local variable, then `i+1` and `i+2` are never-equal.

Otherwise, the compiler conservatively assumes that the expressions are not never-equal.

# 3 Classes and Interfaces

DPJ classes and interfaces are identical to classes and interfaces in Java, with the following additional features:

- New members called *class region name declarations* (§ 3.1) declare region names at class or interface scope that can be associated with class fields (§ 3.2).
- Methods have several new features to support regions and effects (§ 3.3).
- Classes, interfaces, and methods support *region parameters*, so that different objects of the same class or interface, and different invocations of the same method, can use different regions (§ 3.4).
- A new kind of class, called an *array class* (§ 3.5), provides a convenient way to specify array types, particularly ones with type and/or RPL parameters.

Throughout this section, we use the term "class" with the understanding that the concepts apply identically to interfaces, unless otherwise noted.

## 3.1 Class Region Name Declarations

A *class region name declaration* may appear as a class or interface member. A class region name declaration consists of the keyword `region` followed by one or more identifiers separated by commas. For example, the following declaration declares a region name `Data` within the class `Element`:

```
class Element {
    region Data;
    ...
}
```

Class region name declarations function like static class members in Java (though the keyword `static` need not be used with a class region declaration — if it is, it has no effect). In particular, subject to visibilty restrictions, other packages and classes can refer to the declared names by prepending the proper package and class qualifiers. For example, outside of class `Element`, region `Data` could be referred to as `Element.Data`.

As with Java fields and methods, the programmer may control the access to the declared names with the qualifiers `public`, `private`, or `protected`. For example, the following code declares two regions that can be referred to from anywhere in the program:

```
class Node {
    public region Left, Right;
    ...
}
```

Class region name declarations are available for use in RPLs (§ 2) in the scope where they are visible. A class region name standing alone is a particular case of an RPL (see § 2.1.2).

## 3.2 Field RPL Specifiers

Every class field in DPJ resides in a *region*, named with a region path list (§ 2). This is a fundamental aspect of DPJ versus plain Java; it allows the specification and checking of effects (§ 5). There are two ways to specify the RPL of a field: (1) with an explicit *field RPL specifier*; and (2) by using the *default field RPL*.

**Explicit field RPL specifiers:** A field RPL specifier has the following form:

`in` *rpl*

*rpl* is an RPL, as defined in § 2. If the field RPL specifier is present, then it must appear immediately after the field name (and before the field initializer expression, if there is one). For example:

```
class FieldRPLSpecifiers {
    region X, Y;              // X and Y name regions
    int x in X;               // field x is in region X
    boolean y in Y = false;   // field y is in region Y
}
```

Notice that because a class region name declaration functions like a static class member (§ 3.1), it creates a single region name *per class*, not per object. Therefore, a field RPL specifier such as `int x in X` assigns the same region `FieldRPLSpecifiers.X` to the field of *every object instance* of class `FieldRPLSpecifiers` created at runtime. § 3.4 explains how to use *region parameters* to assign different regions to the fields of different objects of the same class.

*Final fields* Effects on `final` fields are ignored (§ 5.4), because their values never change after initialization. Therefore, field RPL specifiers are not meaningful for `final` fields. If a field is declared `final`, then an RPL specifier may be given for the field; but if so, it is ignored.

**Default field RPL:** If a field has no explicit RPL specifier, then its RPL is `Root`. The name `Root` is always in scope (§ 2.1.1). For example, in the following code, field `x` is in region `Root`:

```
class DefaultRegionSpecifier {
    // Equivalent to int x in Root = 5
    int x = 5;
}
```

## 3.3 Methods

DPJ adds the following features to Java methods:

- Every method (§ 3.3.1) and constructor (§ 3.3.2) summarizes its effects, either explicitly or through a default effect summary.
- The programmer may declare and use *local region names* inside a method body (§ 3.3.3), to express effects that do not escape the method scope.
- Methods may be marked `commutative` to indicate that their effects commute, even though they have conflicting reads and writes (§ 3.3.4).

### 3.3.1 Effect Summaries (Non-Constructor Methods)

Every DPJ method must summarize its effects. The compiler uses the summaries to check noninterference of effect (§ 5.7). This section discusses effect summaries for non-constructor methods. § 3.3.2 describes effects summaries for constructors. There are two ways to summarize a method's effects: (1) with an explicit method effect summary; or (2) with the default method effect summary.

**Explicit method effect summaries:** An explicit method effect summary appears immediately after the method's value parameters and before the `throws` clause, if any. It has the form given in § 5.2. For example:

```
class Summaries {
  region X, Y;
  int x in X;
  int y in Y;
  // pureMethod has no effect on the heap
  int pureMethod(int y) pure { return y+1; }
  // throwsMethod reads X and throws an exception
  void throwsMethod() reads X throws Exception {
    if (x != 0) throw new Exception();
  }
  // readWriteMethod reads Y and writes X
  void readWriteMethod() reads Y writes X {
```

```
    x = y;
  }
}
```

A method effect summary must represent all the effects of the method body. More precisely, the actual method effects must be a *subeffect* (§ 5.6) of the summarized effects. The actual effects are computed as described in § 5.4, *except* that any local effect (§ 5.3) can be omitted from the summary. The representation can be conservative, i.e., it is permissible to include effects in the summary that can never occur in executing the method. But if any effect of any possible execution of the method is omitted from the summary, it is a compile-time error.

For example:

```
class MoreSummaries {
  region X;
  int x in X;
  // OK, summary is conservative
  int readsMethod() writes X {
    // Write effects cover reads
    return x;
  }
  // Error!  Read effect must be reported
  int pureMethod() pure {
    return x;
  }
}
```

A method effect summary must also represent all the effects of any method that overrides it. If it does not, then there is a compile-time error. To see why, consider the following classes:

```
class SuperClass {
  region X;
  int x in X;
  void method(int x) writes X {
    this.x = x;
  }
}
class SubClass extends SuperClass {
  // Compile-time error:  pure does not cover writes X
  void method(int x) pure {
    // Do nothing
  }
}
```

The method `method` defined in `SubClass` overrides the method `method` defined in `SuperClass`. There is an error in the code, because the effect `pure` of the subclass method does not cover the effect `writes X` of the superclass method. While this may look innocuous (after all, the `SubClass` version of method really has no effect!), suppose we allowed the code given above and then wrote the following method:

```
void callMethod(SubClass subClass) pure {
  subClass.method();
}
```

Based on the previous code, this looks fine: we are calling `subClass.method`, which has effect `pure`. But does it? Because of polymorphic dispatch, the *runtime* type of the object bound to `subClass` could be `SuperClass`. And in that case, the call to `method` would invoke the *superclass* version of `method`, which has the effect `writes X`. The write to region `X` would be "hidden" by the polymorphic dispatch. To prevent this kind of hiding, DPJ requires that superclass effects cover subclass effects. Notice that if we change the effect summary in the `SubClass` version of `method` to `writes X`, as DPJ requires, then this problem goes away.

**Default effect summary:** Any DPJ method may be written with no explicit method effect summary. In particular, an ordinary Java method is always a valid DPJ method.

If a method lacks an explicit effect summary, then the compiler assigns it the default summary `writes Root:*`. This is the most conservative possible effect summary; it says that the method may read or write to any globally-visible memory location. The default effect summary is always valid, because it covers all possible effects of any method. For example, the following code is valid:

```
class DefaultEffectSummary {
    // Equivalent to 'void method() writes Root:*'
    // OK, because effects may be overreported
    void defaultSummary() {
        // No actual effect!
    }
    void parallel() {
        // Reports interference, even though defaultSummary has no
        // actual effect!
        cobegin {
           defaultSummary();
           defaultSummary();
        }
    }
}
```

In general, the default effect summary is too coarse-grained for methods that are called either directly or indirectly inside a parallel task: as shown in the example, it causes DPJ to detect and warn about interfering writes, even if the method's actual effects are not interfering. However, for methods that are only ever used in sequential parts of the program, the default effect summary saves effort, because the effects of those methods are not important.

### 3.3.2 Effect Summaries (Constructors)

Constructors are special methods; therefore they must also summarize their effects. All the rules given in § 3.3.1 apply, with one important exception: the effect summary of a constructor does *not* have to report any initialization effects on fields of the object being constructed. For example, the following code is legal:

```
class ConstructorExample {
    region X, Y;
    int x in X, y in Y;
    // Effect 'pure' is valid, because initialization effects
    // on x and y don't have to be reported
    ConstructorExample(int x, int y) pure {
        this.x = x;
        this.y = y;
    }
}
```

This works because the DPJ type and effect system guarantees that if one parallel task calls a constructor, then the fields of a constructed object are never read by any other parallel task until the first task is finished. In particular, that means that no other parallel task can ever see object fields in an uninitialized state. So there can be no interference due to initialization effects of constructors.

However, if we wrote the same initializer as a non-constructor method, then the rules in § 3.3.1 would apply, and we would have to write the effects:

```
class NonConstructorExample {
    region X, Y;
    int x in X, y in Y;
    // Effect 'pure' would cause a compile error here
```

```
    static void initialize(int x, int y) writes X, Y {
        this.x = x;
        this.y = y;
    }
}
```

### 3.3.3 Local Region Name Declarations

A region name declaration may appear as a statement in the body of a method. This kind of declaration is called a *local region name declaration*. Like a class region name declaration (§ 3.1), a local region name declaration consists of the keyword `region` followed by a comma-separated list of identifiers. The declared names are available for use in RPLs (§ 2) in the scope of the enclosing statement block. In particular, a local region name standing alone is a valid RPL (§ 2.1.4).

For example, the following code declares region names `A` and `B` that are available for use as region names in the scope of method `localRegionNames`:

```
void localRegionNames() {
    region A, B;
    ...
}
```

Because of their limited scope, local names (and RPLs constructed from them) cannot appear in field RPL specifiers (§ 3.2). They can be used *only* in arguments to class or method region parameters (§ 3.4). Their purpose is to indicate that the effects of operating on the class object, or invoking the method, are local to the enclosing method, and need not to be reported in the effect summary of the enclosing method or constructor (§§ 3.3.1, 3.3.2). This technique is called *effect masking*. For example:

```
class EffectMasking<region R> {
  int x in R;
  // method has no globally-visible effect
  void method() pure {
    // Declare region r local to method
    region r;
    // Use local region to create new EffectMasking object
    // 'masking' cannot escape method
    EffectMasking<r> masking = new EffectMasking<r>();
    // Effect 'writes r' is masked from callees
    masking.x = 0;
  }
}
```

In this example the masked effect is somewhat useless. But there are plenty of realistic cases where objects are created and assigned to temporarily to support some computation, then thrown away because only the final result is needed by the callee. By creating the temporary objects with local regions, the callee can reduce its effect signature, minimizing potential interference and making it more useful inside parallel tasks. See *The Deterministic Parallel Java Tutorial* for more details.

Like Java local variables, a local region name declared in a statement block is in scope only in that block; its scope ends when the block ends. For example, the following code would cause a compile error, because the name `OutOfScopeRegion` is not in scope where it is used:

```
class ScopeExample<region R> {
    void method() {
        {
            region OutOfScopeRegion;
            // Scope ends here
        }
```

```
        // Error:  OutOfScopeRegion is no longer in scope
        ScopeExample<OutOfScopeRegion> x = null;
    }
}
```

### 3.3.4 Commutative Methods

The keyword `commutative` may appear as a method qualifier, before the return type and before the type and/or region parameters of the method, if any:

```
commutative int m(...) { ... }
```

The `commutative` qualifier is a programmer-specified guarantee that any invocation of the method commutes with itself. It is typically used for commutative operations on concurrent data structures such as counter updates, set inserts, histograms, and reductions. These operations write to shared data in a way that "looks interfering" to the DPJ effect system, but due to the semantics of the data structure (which the effect system does not know about), still preserves determinism. For example, here is a use of the `commutative` qualifier to write a simple counter class:

```
class Counter<region R> {
  private int count in R;
  void clear() writes R { count = 0; }
  commutative synchronized void increment() writes R {
    ++count;
  }
  int getCount() reads R { return count; }
}
```

When a method *m* is labeled `commutative`, the DPJ compiler treats the effects of multiple invocations of *m* as noninterfering, even if the read and write effects by themselves, without the `commutative`, would be interfering. For example, in the case of the counter class, concurrent invocations of `increment` have interfering writes to region `R`. However, the `commutative` annotation tells the DPJ compiler to "ignore" that interference. So, for example, the following code compiles with no errors or warnings:

```
Counter<Root> counter = new Counter<Root>();
foreach(int i in 0, 10)
  counter.increment();
```

Note that the `commutative` qualifier *only* tells the DPJ compiler that it is safe to ignore interference. It does *not* introduce any special synchronization or other concurrency control. In particular, it is the programmer's responsibility when using `commutative` to ensure two things:

1. The method is properly synchronized so that concurrent accesses to the method behave as though they have occurred in sequence (i.e., the accesses have *serializable* semantics). In the example above, this property is enforced by making method `increment` `synchronized`. Without the `synchronized` keyword, concurrent invocations of `increment` would have a read-modify-write race, and some of the updates could be lost.

2. The method behaves so that either order of a pair of invocations produces the same result. In the example above, this property holds because incrementing a counter twice, in either order, has the same result (i.e., the final counter value is 2 more than the starting value).

Typically, in user code, the `commutative` annotation is used for simple commutative read-modify-write operations like the one illustrated above. More complicated operations, e.g. set or tree inserts, where the commutativity property is more subtle to verify, would typically be provided by library or framework code. Only very skilled programmers should attempt to "roll their own" implementation of such commutative operations, as the potential for subtle bugs is high, and the DPJ effect system provides no assistance in checking for such bugs.

## 3.4 Region Parameters

DPJ extends Java's generic type parameters by allowing *region parameters* in class and method definitions. Class region parameters become bound to actual regions when the class is instantiated to a type (§ 4.1.1). Method region parameters become bound to actual regions when the method is invoked (§ 3.4.2).

### 3.4.1 Class Region Parameters

**Declaring class region parameters:** As in ordinary generic Java, the class parameters are given as a comma-separated list of identifiers enclosed in angle brackets (<>) immediately after the class name. The new features are as follows:

- Both type and region parameters may appear. If both appear, all the region parameters must follow the type parameters.
- The first region parameter *must* be preceded by the keyword `region`. Any of the other region parameters *may* be preceded by the keyword `region`.
- Any of the type parameters *may* be preceded by the keyword `type`.

The keyword `type` is provided for convenience in distinguishing type from region parameters; but to preserve compatibility with ordinary Java syntax, it is never required.

For example, the following are valid class declarations:

```
// Type parameter T
class A<T> {}
class B<type T> {}
// Region parameter R
class C<region R> {}
// Type parameter T, region parameter R
class D<T, region R> {}
class E<type T, region R> {}
// Type parameters T1, T2, region parameters R1, R2
class F<type T1, T2, region R1, R2> {}
```

**Using class region parameters:** A class region parameter is available for use in RPLs (§ 2.3) within the class definition. In particular, a class region parameter standing alone is a valid RPL. For example, the following code declares a class `ParamInField` with a region parameter `R` and uses `R` in a field RPL specifier (§ 3.2):

```
class ParamInField<region R> {
    public int x in R;
}
```

This code says that the region of field `x` of an object instantiated from the `ParamInField` class is given by the RPL provided as an argument to `R` in the type of the object. For example, this code creates a fresh `ParamInField` object called `p`, such that the field `p.x` is in region `Root`:

```
ParamInField<Root> p = new ParamInField<Root>();
```

As another example of using parameters, the following code declares a class region parameter `R` and uses it to instantiate a type (§ 4.1.1):

```
class ParamInType<region R> {
    ParamInType<R> x in R;
}
```

And this code creates a fresh `ParamInType` object whose field `x` has type `ParamInType<Root>`:

```
ParamInType<Root> p = new ParamInType<Root>();
```

This pattern is very useful for creating chains or graphs of objects, all in the same region. For example, without the region parameter `R`, there would be no way to get the object pointed to by `x` in the same region as the parent object. The RPL specifier `x in R` is insufficient, because that says only that the field `x` is in `R`, not the object that it points to. To get both the field and the object in the same region, you have to write

```
ParamInType<R> x in R
```

as shown. Moreover, DPJ provides the flexibility to assign different regions to the field pointing to the object and the object itself, for example:

```
ParamInType<r1> x in r2
```

This allows for finer-grain partitioning of memory than if the field and the object always had the same region.

As in generic Java, a new declaration of a parameter named `R` hides any parameter named `R` that is already in scope. For example, the following two definitions of the class `Hiding` are equivalent:

```
// First version
class Hiding<region R1> {
    class Inner<region R2> {
        Hiding<R2> x;
    }
}
// Second version
class Hiding<region R> {
    class Inner<region R> {
        Hiding<R> x;
    }
}
```

In both versions we are declaring a parameter for the outer class, declaring a parameter for the inner class, and using the inner class parameter to instantiate a type inside the inner class. In the first version we have given distinct names to the two parameters; while in the second version we have used the same name.

Also as with Java generic type parameters, a class region parameter may not be used in a static context. This is because DPJ region information (like Java generic information) is *erased* during the compilation process. In particular, only one set of code is generated for all types instantiating a given class. If region parameters could be used in a static context, then different code would have to be generated for each different type (as with C++ templates). For example, the following code is invalid, because it attemps to use region `R` in a static context:

```
class Outer<region R> {
    public static class Inner {
        // Not allowed!
        public int x in R;
    }
}
```

Fortunately, this limitation is easy to work around. The standard way (again, as with Java generics) is to introduce a new parameter for the inner class, and write `Inner<R1>` to place `x` in region `R1`:

```
class Outer<region R1> {
    public static class Inner<region R2> {
        public int x in R2;
    }
}
```

### 3.4.2 Method Region Parameters

**Declaring method region parameters:** As in ordinary generic Java, the method region parameters are given as a comma-separated list of identifiers enclosed in angle brackets (<>) immediately before the method

return type (for non-constructor methods) or class name (for constructors). A method parameter declaration has the same form as a class parameter declaration (§ 3.4.1). For example, the following code defines a method with type parameter `T` and region parameter `R`:

```
interface MethodParams {
    public <type T, region R> void method();
}
```

As for class parameters, the `type` keyword is always optional, and the `region` keyword is optional except as to the first region parameter.

**Using method region parameters:** A method region parameter is available for use in RPLs (§ 2.3) within the method definition. In particular, a method region parameter standing alone is a valid RPL. For example, the following code declares a method `method` with a region parameter `R` and uses `R` in the formal (value) parameter and return type of the method:

```
interface MethodParamUses<region R> {
    <region R1>void method(MethodParamUses<R1> arg);
}
```

As in generic Java, a new declaration of a parameter named `R` hides any parameter named `R` that is already in scope. For example, the following code is equivalent to the code above:

```
interface MethodParamUses<region R> {
    <region R>void method(MethodParamUses<R> arg);
}
```

**Invoking methods defined with region parameters:** Subject to the restrictions stated below, RPL arguments to method parameters may be provided explicitly by the programmer, or they may be inferred from the types of the method arguments. In either case, the arguments must obey any disjointness constraints on the method's region parameters (§ 3.4.3).

*Explicit RPL arguments (fully specified):* If the RPL arguments to the method are fully specified (i.e., they contain no `*` or `?`), then the programmer may supply the RPL arguments explicitly, using an extension of the Java syntax for generic method arguments. For this form, as for Java generic methods, there must be an explicit qualifier and a dot preceding the method name; and the arguments must appear in angle brackets after the dot, and before the method name. For example:

`this.<`*args*`>`*meth-name*`()`

Here *args* are type and/or RPL arguments (defined below), and *meth-name* is the name of the method being invoked. Notice that to invoke a method defined in the enclosing class with explicit RPL arguments, `this` (or the class name, for a `static` method) must be used as a qualifier.

The new DPJ features are as follows:

- Both type and region parameters may appear. The number of type and RPL arguments must exactly match the number of type and region parameters (so all type arguments come first, because that is true for the parameters).
- The first region parameter *must* be preceded by the keyword `region`. Any of the other region parameters *may* be preceded by the keyword `region`.
- Any of the type parameters *may* be preceded by the keyword `type`.

The following code example illustrates the use of explicit RPL arguments to a method invocation:

```
abstract class ExplicitArgs<region R> {
    abstract <type T, region R>T callee(ExplicitArgs<R> param);
    ExplicitArgs<Root> caller() {
        ExplicitArgs<Root> arg = new ExplicitArgs<Root>();
        return this.<ExplicitArgs<Root>,Root>callee(arg);
```

```
    }
}
```

The abstract method `callee` is generic in type `T` and region `R`. It takes an `ExplicitArgs<R>` object as a value argument and returns type `T`. In `caller`, we create a new `ExplicitArgs<Root>` and pass it to `callee`. We pass `ExplicitArgs<Root>` as the type argument to `T` and `Root` as the RPL argument to `R`, using explicit RPL arguments. Notice that the type of `arg` matches the type of `param`, after substituting `Root` for `R` in `param`. If the types did not match, there would be a compile error. See § 4.3.3 for more information about how typing works for method invocations in the presence of region parameters.

Note that the keyword `region` must be present to identify the first RPL argument in the list, unlike the case of RPL arguments to classes (§ 4.1.1), where the `region` keyword is optional. The reason for this rule is Java's method overloading: because multiple methods can be declared with the same name but different parameters, information about which arguments are types and which are regions is not always available from the method name, as it is from the class name.

*Explicit RPL arguments (partially specified):* Explicit RPL arguments to a method invocation may contain partially specified RPLs (§ 2.4). However, if a partially specified RPL is bound to a method region parameter `R`, then `R` may not appear in any of the types of the method arguments. For example, the following code causes a compile error:

```
abstract class PartiallySpecifiedBad<region R> {
    region r1, r2;
    abstract <region R>void callee(PartiallySpecifiedBad<R> param1,
                                   PartiallySpecifiedBad<R> param2);
    void caller() {
        PartiallySpecifiedBad<r1> arg1 = new PartiallySpecifiedBad<r1>();
        PartiallySpecifiedBad<r2> arg2 = new PartiallySpecifiedBad<r2>();
        return this.<*>callee(arg1,arg2);
    }
}
```

If this code were allowed, then both `r1` and `r2` would be bound to `R` inside the scope of `callee`. This would violate a key invariant of the system, namely that in any scope a parameter `R` always refers consistently to a single region. Technically, the compiler avoids this problem by *capturing* any fully specified RPLs provided as an explicit argument to a method RPL parameter (see § 4.3.4). The capture operation declares a fresh parameter which stands in for the runtime region represented by the `*`.

In practice, this means that explicit partially-specified arguments to method RPLs are rarely used. Instead, an inferred argument would be used (see immediately below). However, an explicit partially-specified argument may be used to specify an effect. For example, this code is legal:

```
abstract class PartiallySpecifiedOK<region R> {
    abstract <region R>void callee() writes R;
    void caller() writes * {
        // Effect is writes *
        return this.<*>callee(arg1,arg2);
    }
}
```

*Inferred RPL arguments:* As in generic Java, a method with type and/or region parameters may be written without any explicit generic arguments. In this case the compiler will attempt to infer the type and/or region arguments from the types of the value arguments supplied to the method. For example, the `dpjc` compiler accepts the following code, because it is able to infer from the type `InferredArguments<Root>` of `arg` that the argument to `R` is `Root`:

```
abstract class InferredArguments<region R> {
    abstract <region R>void callee(InferredArguments<R> param);
    void caller() {
        InferredArguments<Root> arg = new InferredArguments<Root>();
```

```
        callee(arg);
    }
}
```

This code is equivalent to the following:

```
abstract class InferredArguments<region R> {
    abstract <region R>void callee(InferredArguments<R> param);
    void caller() {
        InferredArguments<Root> arg = new InferredArguments<Root>();
        this.<Root>callee(arg);
    }
}
```

While the compiler can infer region arguments to methods in many cases, sometimes it cannot, either because the inference algorithm is insufficiently powerful, or because the information is simply not available from the types of the value parameters. The compiler uses `Root:*` as the argument to any region parameters in the method that it cannot infer. For example, the compiler would infer `Root:*` as the argument to `R` in the following code:

```
abstract class InferredArguments<region R> {
    abstract <region R>void callee() writes R;
    void caller() {
        // There are no value arguments, so there is no way to infer
        // the argument to R!  This is equivalent to this.<Root:*>callee();
        callee();
    }
}
```

Finally, because of the restriction on partially-specified RPLs in explicit arguments mentioned above, it is possible to write certain method invocations using inferred arguments that it is *not* possible to write using explicit arguments. For example:

```
abstract class InferredArguments<region R> {
    abstract <region R>void callee(InferredArguments<R> param) writes R;
    void caller() {
        InferredArguments<*> arg = new InferredArguments<Root>();
        callee(arg);
    }
}
```

Here the compiler infers that at the invocation of `callee` a fresh region parameter should be bound to both the parameter `R` in the method and the parameter `R` in the type.

Notice that because the invocation of `callee` binds `*` to `R`, and `R` appears in the type of `param`, it would not be legal to provide the RPL argument explicitly:

```
this.<*>callee(arg); // illegal!
```

That is because this code provides no assurance that the region bound to the method parameter is the same as the region bound to the type of `arg`. In DPJ, region inference is the only way to declare a fresh RPL parameter and bind it to multiple region arguments (this is sometimes called "opening an existential"). Generic Java has a similar mechanism for handling wildcard types in method arguments.

### 3.4.3 Disjointness Constraints

It is sometimes necessary to require that two or more region parameters (or a region parameter and some other region) be disjoint. For example, the following code can have a data race unless regions `R1` and `R2` are bound to disjoint regions:

```
class Unsafe<region R> {
    int x in R;
    <region R1, R2>void method(Unsafe<R1> o1, Unsafe<R2> o2) {
        cobegin {
            ++o1.x; // writes R1
            ++o2.x; // writes R2
        }
    }
}
```

To support this reasoning, DPJ provides optional *disjointness constraints* for region parameters. If used, the constraints must appear after the region parameters, and be separated from them by a vertical bar:

`<region R1, R2, ... |` *constraints* `>`

*constraints* is a comma-separated list of constraints, where a constraint has the form *rpl1* `#` *rpl2*, and *rpl1* and *rpl2* are valid RPLs (§ 2). The constraint states that *rpl1* and *rpl2* are disjoint regions (§ 2.6.4). For class region parameters, the requirement is enforced when the class is instantiated to a type by substituting RPLs for parameters (§ 4.1.1). For method region parameters, the requirement is enforced when RPL arguments are provided to an invocation of the method (§ 3.4.2).

For example, the code above could be rewritten as follows to make it safe:

```
class Safe<region R> {
    int x in R;
    <region R1, R2 | R1 # R2> void(Safe<R1> o1, Safe<R2> o2) {
        cobegin {
            ++o1.x; // writes R1
            ++o2.x; // writes R2
        }
    }
}
```

There is no interference between the statements of the `cobegin`, because `R1` and `R2` are guaranteed to be disjoint regions. Therefore, the effects of `++o1.x` and `++o2.x` are to disjoint memory locations.

We can also use constraints to make parameters disjoint from region names, not just other parameters. For example, we can write the following:

```
class Safe2<region R> {
    region r;
    int x in R;
    <region R | R # r>void(Safe2<R> o1, Safe2<r> o2) {
        cobegin {
            ++o1.x; // writes R
            ++o2.x; // writes r
        }
    }
}
```

This technique is useful when methods need to operate disjointly in parallel on global data and data passed in as an an argument. Of course, the region of the global data could also be passed in as an RPL argument, but using the global region directly in the constraint saves writing parameters and arguments.

It is an error to write a constraint that cannot be satisfied, because the RPLs given cannot be disjoint. For instance, this class declaration generates a compile error:

```
class BadConstraint<region R | R # R> {}
```

It is also an error to provide RPL arguments to a class or method that do not satisfy the constraints. For example:

```
abstract class BadArgs<region R1, R2 | R1 # R2> {
    // Error:  Instantiating BadArgs with R1=Root, R2=Root
    BadArgs<Root, Root> x;
    abstract <region R3, R4 | R3 # R4>void callee();
    // Error:  Invoking callee with R3=Root, R4=Root
    void caller() {
        this.<Root,Root>callee();
    }
}
```

`BadArgs<Root, Root>` is an invalid type, because the RPLs supplied as arguments to `R1` and `R2` are not disjoint, as required by the definition of class `BadArgs`. Similarly, `this.<Root,Root>callee()` is an invalid invocation of `callee`, because the RPLs supplied as arguments to `R3` and `R4` are not disjoint, as required by the definition of method `callee`.

## 3.5 Array Classes

DPJ v1.0 added region parameters directly to Java's syntax for array types. This is fine for simple cases, but it can quickly get unwieldy. Therefore, DPJ v1.1 adds a new feature called an *array class*. Syntactically an array class is similar to an ordinary Java class. However, instead of specifying an ordinary Java class object with fields and methods, it specifies a Java array object by assigning a type and region to each cell of the array.

### 3.5.1 Defining Array Classes

An array class definition is like an ordinary class definition with a single field. The differences are: (1) the definition begins with the keyword `arrayclass` instead of `class`; (2) the "field" has no name (since it actually defines the type of an array cell); and (3) in addition to defining the cell the class definition may declare regions, but it may have no other members.

For example, the following code defines an array whose cells have type `int` and are located in region `Root`:

```
arrayclass ArrayInt { int; }
```

And the following code declares a region `r` and defines an array whose cells have type `int` and are located in region `ArrayInt2.r`:

```
arrayclass ArrayInt2 {
    region r;
    int in r;
}
```

Type and region parameters work just as for ordinary Java classes (§ 3.4). For example, the following code declares an array class with one type parameter `T`, one region parameter `R`, and cells of type `T` in region `R`:

```
arrayclass Array<type T, region R> {
    T in R;
}
```

The following code is illegal, because it attempts to define an instance method for an array class, which is not allowed:

```
arrayclass Illegal {
    int;
    int m() { return 0; }
}
```

The following code is also illegal, because it doesn't provide a type for the array cells:

```
arrayclass Illegal2 {
    region r;
}
```

### 3.5.2 Creating and Using Array Class Objects

Every array class has an implicit constructor that takes a single integer length argument. Calling the constructor creates a new array with the specified type, region, and length. For example, using the array classes defined in the previous section, we could write the following code:

```
// Create an array of 10 int in Root
ArrayInt a1 = new ArrayInt(10);
// Create an array of 10 int in ArrayInt2.r
ArrayInt2 a2 = new ArrayInt2(10);
// Create an array of 10 Integer in r
region r;
Array<Integer,r> a3 = new Array<Integer,r>(10);
```

In each case, the effect of the constructor call is to create an ordinary Java array. For example, the compiler will translate `new ArrayInt(10)` to `new int[10]` before generating code.

Once created, an array class object works just like an ordinary Java array object: its fields can be read and written with the index operator `[...]`, it has an implicit `length` field, and it works with the "enhanced for" construct of Java 5. For example, the following code creates and populates array of 10 `int` values, and then prints them out:

```
ArrayInt a = new ArrayInt(10):
for (int i = 0; i < a.length; ++i)
    a[i] = i;
for (int i : a)
    System.out.println(i);
```

### 3.5.3 Index-Parameterized Array Classes

For parallel computations on arrays, it is often necessary to place different array cells in different regions, so they can be updated in parallel. In DPJ v1.1, you do this by writing an array region `[index]` in the scope of the array class definition, where `index` is an implicit field that is in scope in every array class definition (DPJ v1.0 used a different syntax). The region `[index]` stands in for the index region associated with each cell. For example:

```
// Array of int such that cell i is in region [i]
arrayclass IPArrayInt {
    int in [index];
}
```

Now we can do disjoint initialization of an array using a parallel `foreach` loop (see § 6.2):

```
IPArrayInt arr = new IPArrayInt(N);
foreach (int i in 0, N)
    // Write to each distinct arr[i] in parallel
    arr[i] = i;
```

The compiler knows this is safe, because the writes to arr[i] touch a different region for each iteration of the parallel loop.

Index-parameterized arrays also support updates through references to different objects, by using `[index]` in the RPL argument of the element type of the array. Every object stored in a distinct array cell gets its own region, parameterized by the index of the cell. For example, suppose we have the following simple class definitions:

```
class Data<region R> {
    int field in R;
    static arrayclass Array<region R> {
        Data<R:[index]> in R:[index];
```

```
    }
}
```

Then we can create the following index-parameterized array:

```
Data.Array arr = new Data.Array(N);
```

The type of `arr` is an array such that (1) cell `i` of the array is in region `[i]`; and (2) cell `i` of the array has type `Data<[i]>`. Now we can do disjoint initialization of the array, as before:

```
foreach (int i in 0, N)
    arr[i] = new Data<[i]>();
```

Notice that the type `Data<[i]>` on the left-hand side of the assignment matches the type of `arr[i]`, as it must; see § 4.3.2.

We can also update the objects disjointly through the references in the array:

```
foreach (int i in 0, N)
    // Effect is 'writes [i]'
    ++arr[i].field;
```

This is a very useful pattern in shared-memory parallel programming. See *The Deterministic Parallel Java Tutorial* for more realistic examples.

### 3.5.4 Arrays of Arrays

Array class definitions may be composed to define arrays of arrays. For example, to define an array class equivalent to the ordinary Java array type `int[][]`, one could do this:

```
arrayclass MatrixInt {
    ArrayInt;
}
arrayclass ArrayInt {
    int;
}
```

Then we could write the following code:

```
// Create array of 10 ArrayInt
MatrixInt arr = new MatrixInt(10);
// Fill it in with 10 ArrayInts
for (int i = 0; i < 10; ++i)
    arr[i] = new ArrayInt(10);
// Initialize the ArrayInts
for (int i = 0; i < 10; ++i)
    for (int j = 0; j < 10; ++j)
        arr[i][j] = 10 * i + j;
```

Of course if we decompose the arrays into regions, then we can update them in parallel. Here is another example that uses index-parameterized arrays to do just that:

```
class MatrixExample {
    static arrayclass MatrixInt<region R> {
        ArrayInt<R:[index]> in R:[index];
    }
    static arrayclass ArrayInt<region R> {
        int in R:[index];
    }
    public static final int N = 10;

    public static void main(String[] args) {
```

```
        region r;
        MatrixInt<r> matrix = new MatrixInt<r>(N);
        foreach (int i in 0, N) {
            // Effect is 'writes [i]'
            matrix[i] = new ArrayInt<r:[i]>(N);
            foreach (int j in 0, N) {
                // Effect is 'writes [i]:[j]'
                matrix[i][j] = N*i+j;
            }
        }
    }
}
```

Because of the region decomposition, the compiler can use the effects to prove that the parallel tasks are noninterfering. For the details of how the compiler would check this example, see § 5.4 (computing effects of statements) and § 5.7 (noninterfering effects).

### 3.5.5 Generic Arrays

Unfortunately, Java does not let you create arrays of class types that have generic parameters without an explicit cast. For example, the following code will not work properly:

```
class Data<type T> {
    T field;
    arrayclass Array {
        Data<Integer>;
    }
    void m() {
        Array a = new Array(10);
    }
}
```

This code passes the DPJ compiler, but it will not compile correctly to Java bytecode, because the generated code causes a "generic array creation" error.

To get around this limitation, you need to use a cast:

```
class Data<type T> {
    T field;
    arrayclass Array {
        Data<Integer>;
    }
    void m() {
        // Rewrite of the last line above so it compiles
        Array a = (Array) ((Object) new Object[10]);
    }
}
```

This code is ugly, but it works. Also, the ugliness is localized to the point of array creation. Once the array is created and assigned, everything else works as it should.

# 4 Types

DPJ extends the Java type system by adding RPLs (§ 2) to class and array types. The RPLs in the types support the effect system: different class and array objects can be created with different RPLs, and the compiler can use the RPLs to infer the effect of an operation on the object (i.e., accessing a class field or array cell, or invoking a method).

This section discusses DPJ's extensions to the Java type system: § 4.1 discusses class and interface types, § 4.2 discusses array types, and § 4.3 discusses the rules for determining the type of an expression (e.g., what type is returned by calling a method).

## 4.1 Class and Interface Types

This section discusses DPJ's class types (including array classes) and interface types. We will use the term "class" throughout this section, with the understanding that the concepts apply identically to array classes and interfaces, unless otherwise noted. § 4.1.1 discusses the instantiation of a class to a type by supplying RPLs for its parameters. § 4.1.2 explains how the DPJ compiler compares two class types for compatibility, for example in checking an assignment statement. § 4.1.3 discusses casting one type to another in DPJ. § 4.1.4 explains *owner RPLs*, which determine the nesting relation for variable RPLs (§ 2.1.5).

### 4.1.1 Writing Class Types

In DPJ, as in generic Java, a class with parameters defines a family of types, one for each set of arguments to the parameters. However, in addition to type parameters, DPJ classes can have region parameters (§ 3.4.1). When writing a DPJ class type, parameter arguments must be supplied. They can be supplied implicitly or explicitly.

**Implicit parameter arguments:** As in generic Java, a DPJ class with parameters may always be used as a type by just naming the class, without providing any explicit parameter arguments. For example, the following code is valid:

```
class ImplicitArgs<type T, region R> {
    // ImplicitArgs is a valid type here
    ImplicitArgs field = null;
}
```

When the type arguments are omitted for a class with parameters, the following occurs:

- The type parameters, if any, have no argument. That is, the type functions as a *raw type* in generic Java.
- The RPL parameters, if any, become bound to `Root`.

In the example written above, `field` is of class type `ImplicitArgs`, with `Root` bound to `R` and no argument for `T`.

Implicit parameter arguments are useful in two ways. First they help with porting legacy Java code to DPJ. For example, adding a region parameter to a preexisting class will not break any code that uses the class. Second, because region arguments are used to compute the effects on fields of a class (§ 5.4), they are usually not important for code that is never invoked in a parallel context (for example, in a sequential initialization phase). For this code, the programmer can avoid writing the arguments.

**Explicit parameter arguments:** As in Java, if explicit arguments to class parameters are given, then they appear in angle brackets after the class name. The arguments must appear in the same order in which the parameters appear in the class definition; so, in particular, any type arguments must precede any RPL arguments. Any of the type arguments may be preceded by the keyword `type`, and any of the RPL arguments may be preceded by the keyword `region`; but the keywords are optional, as the compiler can infer which are the types and which are the RPLs from the class definition.

For example:

```
class ExplicitArgs<type T, region R> {
    // Instantiate an ExplicitArgs type with T=Object, R=Root
    ExplicitArgs<Object,Root> field;
}
```

This code defines a class `ExplicitArgs` with one type parameter `T` and one RPL parameter `R`. The field `field` has class type `ExplicitArgs` with `T` = `Object` and `R` = `Root`.

The number of type arguments must exactly match the number of type parameters. However, fewer RPL arguments can be given than the number of RPL parameters; any missing RPL arguments (matching the arguments that are there from the left) are implicitly bound to `Root`. For example, in the code above, we could have written the type of field `field` as `ExplicitArgs<Object>`. Here is another example:

```
class ImplicitRPLArg<region R1, R2> {
    region r;
    ImplicitRPLArg<r> field;
}
```

The class `ImplicitRPLArg` has two parameters `R1` and `R2`, but the type `ImplicitRPLArg<r>` of `field` has only one explicit RPL argument, `r`. `r` is bound to the first parameter `R1`, and the second parameter has no argument, so it is bound to `Root`. This type is the same as `ImplicitRPLArg<r,Root>`.

### 4.1.2 Class Type Comparisons

The notion of *type comparison* is important in Java. For example, if class `B` extends class `A`, and variable `x` has type `A`, then you can assign an object of type `B` to `x`; but it is not permissible to assign an object of some type `C` that does not extend `A` (or extend another class that extends `A`, etc.).

DPJ extends Java's type comparison rules to account for the RPL information in the class types. The rules ensure that the actual type of an object always corresponds to the type that appears in a program variable storing a reference to the object. This property in turn allows the compiler to reason soundly about effects by looking at the types of variables. If the "wrong" type of object could be assigned to a variable, then this kind of reasoning would not work.

To state the comparison rules for DPJ, we introduce the concept of the *DPJ erasure* of a type. This is the type obtained by ignoring all RPL parameters and arguments. For example, suppose class `C` has a type parameter `T` and a region parameter `R`. Then the DPJ erasure of `C<Object,Root>` is `C<Object>`.

The following rules determine whether a variable of class type $T_1$ can be assigned to a variable of class type $T_2$ in DPJ.

**Types that instantiate the same class:** $T_1$ may be assigned to $T_2$ if the two types instantiate the same class, the DPJ erasure of $T_1$ may be assigned to the DPJ erasure of $T_2$ in ordinary Java, and the RPL arguments of $T_1$ are included (§ 2.6.3) in the corresponding arguments of $T_2$.

Here are some examples of compatible types that instantiate the same class:

- `C<Root>` may be assigned to `C<Root:*>`, because the DPJ erasures are identical, and `Root:*` includes `Root`.
- `C<Object,Root>` may be assigned to `C<Object,Root:*>` for the same reason.
- `C<C<Object,Root>,Root>` may be assigned to `C<? extends C<Object,Root:*>,Root:*>`. The reasoning here is a bit more complicated. First, look at the DPJ erasures of the types, that is `C<C<Object,Root>>` and `C<? extends C<Object,Root:*>>`. According to the rules for Java generic wildcards, the first type may be assigned to the second if `C<Object,Root>` may be assigned to `C<Object,Root:*>`. To figure that out, we have to apply the rule recursively. First look at the DPJ erasures: assigning `C<Object>` to `C<Object>` is OK. Now look at the RPLs: `Root:*` includes `Root`. So that test checks out. Now look at the RPL arguments in the original types: `Root:*` includes `Root`. So that test checks out as well.

Here are some examples of incompatible types that instantiate the same class, assuming `r1` and `r2` are region names:

- `C<r1>` may not be assigned to `C<r2>`, because `r1` is not included in `r2`.
- `C<Object,r1>` may not be assigned to `C<Object,r2>` for the same reason.
- `C<C<Object,r1>,Root>` may not be assigned to `C<? extends C<Object,r2>,Root` because `C<Object,r1>` may not be assigned to `C<Object,r2>`.

**Types that instantiate different classes:** $T_1$ may be assigned to $T_2$ if $T_2$ is a *direct supertype* or *indirect supertype* of $T_1$.
*Direct supertypes:* $T_2$ is a direct supertype of $T_1$ if a type with $T_2$'s class or interface appears in an `extends` or `implements` clause of $T_1$'s class or interface, and that type is assignable to $T_2$ after substituting arguments for parameters according to $T_1$.

For example:

- Assume class declarations `class B<region R> extends A<R>` and `class A<region R>`. Then `A<Root>` is a direct supertype of `B<Root>`, because `A<R>` appears in the `extends` clause of `B`, and the type obtained by substituting `Root` for `R` from the type `B<Root>` is `A<Root>`.
- With the same assumptions as in (1), `A<Root:*>` is a direct supertype of `B<Root>`, because `A<Root>` is assignable to `A<Root:*>`.
- Assume class declaration `class B<type T, region R> extends A<B<Root>,R>`. Then `A<B<Root>,Root:*>` is a direct supertype of `B<Object,Root>`.

*Indirect supertypes:* $T_2$ is an indirect supertype of $T_1$ if there is a chain of direct supertypes connecting $T_1$ to $T_2$. For example, if we have class declarations `class C<region R> extends B<R>`, `class B<region R> extends A<R>` and `class A<region R>`, then `A<Root>` is an indirect supertype of `C<Root>`, because `A<Root>` is a direct supertype of `B<Root>`, and `B<Root>` is a direct supertype of `C<Root>`.

### 4.1.3 Class Type Casts

One DPJ type may be cast to another if the cast would be allowed for the DPJ erasures of the types (§ 4.1.2) in ordinary Java. For example, assuming a class declared as `class C<region R>` the following cast is legal:

```
C<r1> x = (C<r1>) new C<r2>();
```

This code creates an object of type `C<r2>` on the right-hand side, then casts it to type `C<r1>` before assigning it to a variable `x` of type `C<r1>`. The DPJ erasure of both the `new` expression and the target type is `C`, so the cast is allowed. Without the cast, `C<r2>` is not assignable to `C<r1>`, because `r2` does not include `r1`.

However, the following cast is not legal, assuming a class declared as `class C<type T, region R>`:

```
// Compile error!
C<C<Object,r1>,r1 x =
  (C<C<Object,r1>,r1>) new C<C<Object,r2>,r2>();
```

That is because the DPJ erasure of the type of the `new` expression is `C<Object,r2>`; the DPJ erasure of the target type is `C<Object,r1>`; and these two types are not compatible.

As in ordinary Java, casts allow assignments that the compiler cannot check for correctness. That is, in general for a statement of the form

```
T x = (T) y
```

there is no way to ensure at compile time that the type of the object `y` is really consistent with the type `T` of variable `x`. Further, for efficiency reasons, DPJ has no checks for catching bad assignments at runtime. Therefore, a DPJ program with casts in it is a potentially nondeterministic program! Casts should be used very carefully and only as a last resort, when there is no other way to express the program.

### 4.1.4 Owner RPLs

Owner RPLs define the nesting relationship between `final` local variables used as RPLs (§ 2.1.5) and other RPLs. A variable RPL is nested under the owner RPL of its class type (§ 2.6.2).

If a class has RPL parameters, then the owner RPL of a type instantiating the class is the argument to the first parameter. For example, assuming a class declaration `class C<region R1,R2>`, and region names `r1` and `r2`, the owner RPL of the type `C<r1,r2>` is `r1`.

If a class does not have RPL parameters, then the owner RPL of a type instantiating the class is `Root`. For example, assuming a class declaration `class C<type T>`, the owner RPL of the type `C<Object>` is `Root`.

## 4.2 Array Types

This section discusses DPJ's support for Java-style array types, i.e., types of the form `T[]`. For the most part, array classes (§ 3.5) should be more convenient to use than Java-style array types. However, for backwards compatibility, Java-style array types are still supported.

§ 4.2.1 explains how to use array types in DPJ. § 4.2.2 explains how the DPJ compiler compares two array types for compatibility, for example in checking an assignment statement. § 4.2.3 discusses casting one array type to another in DPJ.

### 4.2.1 Using Array Types

A DPJ array type is the same as an ordinary Java array type, i.e., it has the form `T[]` for some type `T` (which may itself be an array type). The only difference is that `T` may be a class with region parameter arguments, or it may be an array type constructed from such a class.

Creation of array objects via `new` is exactly the same as in Java. For example, `new String[10]` creates a new array of 10 `String` objects.

Access to an array cell via an index operation always touches the region `Root`. If an array in a different region is desired, an array class must be used (§ 3.5).

### 4.2.2 Array Type Comparisons

The following rules determine when $T_1$ may be assigned to $T_2$, where $T_1$ and $T_2$ are both array types.
**Comparing arrays of primitive types:** $T_1$ may be assigned to $T_2$ if the element types of $T_1$ and $T_2$ are both the same primitive type.
**Comparing arrays of class and array types:** $T_1$ may be assigned to $T_2$ if the element types of $T_1$ and $T_2$ are *array-compatible* class or array types. Two class types are array-compatible if the element type of $T_1$ can be assigned to the element type of $T_2$ using the rules in §§ 4.1.2, 4.2.2, but requiring equivalence (§ 2.6.1) instead of inclusion of RPLs. For example:

1. `C<R>[]` may be a assigned to `C<R>[]`, because `C<R>` is array compatible with itself.
2. `C<R>[]` may not be assigned to `C<R:*>[]`, because `C<R>` and `C<R:*>` are not array-compatible.

The extra requirement of array-compatibility is introduced to avoid problems like this:

```
class C<region R> { ... }
region r1, r2;
// Create an array of 10 C<r1>
C<r1>[] arr = new C<r1>[10];
// Not really allowed, but would be without array compatibility
C<*>[] = badArr;
// Inconsistent types!  Assigning C<r2> to a cell of arr
badArr[0] = new C<r2>();
```

### 4.2.3 Array Type Casts

Casts of array types have the same rules as casts of class types (§ 4.1.3). The cast is allowed if it would be allowed for the DPJ erasures of the types in ordinary Java. The DPJ erasure of an array type is computed by taking the erasure of the class type appearing to the left of all brackets. For example, the DPJ erasure of `C<R>[]` is `C[]`.

## 4.3 Typing Expressions

In Java, every expression has a type. For example, if a method returns `int`, then an expression invoking that method has type `int`. (Some statements have types too, but those statements always enclose expressions.) The types allow the compiler to enforce consistency of assignments; for example, to make sure that an `int` is never assigned to a variable of type `String`.

In DPJ, every expression has a type and an effect. § 5.4 discusses the effects of statements and expressions. Here we discuss DPJ's extensions to Java's rules for determinining the type of an expression.

### 4.3.1 Field Access

A field access expression in Java has the general form

*receiver-exp* . *field-name*

where *receiver-exp* is a class name $C$ or expression of class type $C$, and *field-name* is the name of a field of class $C$ (or a superclass of $C$). If a bare field name appears, then the implied receiver is either `C`.*field-name* where `C` is the enclosing class (for a static field), or `this`.*field-name* (for an instance field). Here we give the rules for the general form.

For instance fields (i.e., *receiver-exp* is an expression of class type), the compiler carries out the following steps to determine the type of an expression *receiver-exp* . *field-name*:

1. Determine the type *receiver-type* of *receiver-exp*, using the rules in this section together with the ordinary rules for Java types.
2. Look up the type *field-type* of `field` based on the class $C$ named in *receiver-type*.
3. Compute the *capture* (§ 4.3.4) of *receiver-type* to generate the type *captured-receiver-type*.
4. Make the following subsitutions in *field-type* to generate the answer:
   (a) Substitute the type and RPL arguments of *captured-receiver-type* for the corresponding type and RPL parameters of class $C$.
   (b) If *receiver-exp* is a `final` local variable or `this`, then substitute the variable for `this`. Otherwise, substitute a capture parameter (§ 4.3.4) included in *owner-rpl*:$*$, where *owner-rpl* is the owner RPL of the variable (§ 4.1.4).

For static fields (i.e., *receiver-exp* is a class name), only steps 1 and 2 are required.

Here is some example code for which the capture in step 3 is a no-op, and the capture in step 4(b) isn't needed. § 4.3.4 gives examples involving capture.

```
1  class FieldTypingExample<region R> {
2      region r;
3      FieldTypingExample<R> field1 in R;
4      void FieldTypingExample<r> method1() {
5          return (new FieldTypingExample<r>).field1;
6      }
7      FieldTypingExample<this> field2 in this;
8      void FieldTypingExample<arg>
9        method2(final FieldTypingExample<R> arg) {
10         return arg.field;
11     }
12 }
```

Let's see how to compute the type of the expression in the `return` statement in line 5.

1. The receiver expression is a `new` expression of type `FieldTypingExample<r>`.

2. The field name is `field1`, the class $C$ is `FieldTypingExample`, and the declared type of the field is `FieldTypingExample<R>` (line 3).

3. Nothing to do (see § 4.3.4).

4. Substituting `r` given in the receiver type expression for `R` in the field type, we have that the answer is `FieldTypingExample<r>`.

Now let's see how to compute the type of the expression in the `return` statement in line 10.

1. The receiver expression is `arg`, which has type `FieldTypingExample<R>`.

2. Now the field is `field2`, and its type is `FieldTypingExample<this>` (line 7).

3. Again, nothing to do.

4. `arg` is a `final` local variable (line 9), so we can substitute it for `this` in the type of `field2`. Therefore the answer is `FieldTypingExample<arg>`.

### 4.3.2 Array Access

An array access expression has the form *array-exp*`[`*index-exp*`]`, where *array-exp* is an expression of array class type (§ 4.1) or array type (§ 4.2), and *index-exp* is an expression of integer type.
**Array class type:** If *array-exp* has array class type, then do the following:

1. perform the same conversion as for field access (§ 4.3.1), treating the cell type of the array class as the field type.

2. Substitute *receiver-exp* for `index` in the result.

For example, given the following code

```
class Data<region R> {
    static arrayclass Array<region R> {
        Data<R:[i]> in R:[i];
    }
}
region r;
Data.Array<r> a = new Data.Array<r>(10);
```

The access expression `a[0]` has type `Data<r:[0]>`.
**Array type:** If *array-exp* has array type, then delete one set of brackets to get the type, as in ordinary Java.

### 4.3.3 Method Invocation

**Explicit type and RPL arguments:** To compute the type of a method invocation expression

*receiver-exp*`.<`*type-args*`,`*rpl-args*`>`*method-name*`(`*args*`)`

The compiler carries out the following steps:

1. Determine the type *receiver-type* of *receiver-exp* and the types *arg-types* of *args*, using the rules in this section together with the ordinary rules for Java types.

2. Look up the method in the ordinary Java way, using the method name, *receiver-type*, and *arg-types*. Use the method to find formal value parameter types and return type.

3. Compute the *capture* (§ 4.3.4) of *receiver-type* to generate the type *captured-receiver-type*.

4. Make the following substitutions in each of the formal argument types then check that the actual argument types *arg-types* are assignable (§§ 4.1.2 and 4.2.2) to the formal argument types:

   (a) Substitute the type and RPL arguments of *captured-receiver-type* for the type and RPL arguments of its class.

   (b) Capture *type-args* and *rpl-args* and substitute the captured types and RPLs for the type and RPL parameters of the method.

   (c) For every argument in *args* of a type assignable to `int`, substitute the actual argument expression for the corresponding formal parameter of the method.

   (d) If *receiver-exp* is a `final` local variable or `this`, then substitute the variable for `this`. Otherwise substitute a capture parameter (§ 4.3.4) included in *owner-rpl*`:*`, where *owner-rpl* is the owner RPL of the variable (§ 4.1.4).

5. Make the same substitutions as in step 4 in the return type to compute the answer.

Here is some example code for which the capture in steps 3 and 4(b) are no-ops, and the capture in step 4(d) isn't needed. § 4.3.4 gives examples involving capture.

```
1 abstract class MethodInvocationExample<region R1, R2> {
2     abstract <region R3>
3         MethodInvocationExample<R3, [i]>callee(int i);
4     MethodInvocationExample<Root, [0]>caller() {
5         return this.<Root>callee(0);
6     }
7 }
```

Lines 1–2 define a method `callee` with one RPL parameter `R3` and one formal parameter `int i`. It returns type

`MethodInvocationExample<R3,[i]>`.

Notice that the return type is written in terms of the method parameter (both the RPL and value parameter); these parameters have to be substituted away to generate a meaningful type at the call site.

At the call site (line 5), the compiler does the following.

1. The receiver type is `MethodRegionInvocationExample<R1,R2>`, and the argument type is `int`.
2. The invoked method is `callee`, defined in lines 1–2.
3. Nothing to do (see § 4.3.4).
4. Binding 0 to `int` is OK.
5. The return type is `<MethodInvocationExample<R3,[i]>`. The method arguments are `R3` = `Root` and `i` = 0. Substituting arguments for parameters gives a return type of `MethodInvocationExample<Root,[0]>`.

**Inferred type and RPL arguments:** If there are no explicit type or RPL arguments, the compiler infers them as discussed in § 3.4.2 and proceeds as stated above, using the inferred arguments in step 4(b). As in ordinary Java, if there is no *receiver-exp*, then the implied receiver expression is `this`.

### 4.3.4 Captured Types

The *capture* of a generic type is important in generic Java; it prevents bad assignments through wildcard types. DPJ uses the same concept in connection with partially-specified RPLs, which are a kind of wildcard, as they can stand in for several different regions. To motivate the problem, consider this example:

```
1  class CaptureExample<region R> {
2      CaptureExample<R> field;
3      void method() {
4          region r1, r2;
5          // Create a new CaptureExample<R:r1>
6          CaptureExample<R:r1> ce = new CaptureExample<R:r1>();
7          // Assign it to CaptureExample<R:*>; this is allowed
8          CaptureExample<R:*> ceStar = ce;
9          // This should not be allowed!
10         ceStar.field = new CaptureExample<R:r2>();
11     }
12 }
```

The assignment in line 10 is a problem: the *actual* type of the object stored in `ceStar` is `CaptureExample<R:r1>`. That's because the assignment in line 8 didn't change the type of the object; it just assigned the same object to a different reference with a partially specified type. So the actual type of the `field` field of that object is `CaptureExample<R:r1>`. That means it can't hold a `CaptureExample<R:r2>`; that would violate the consistency of typing at runtime. However, the type of the receiver expression `ceStar` is `CaptureExample<R:*>`. If we computed the type of `ceStar.field` by just substituting the actual for formal arguments in the type of the receiver expression, we would get `CaptureExample<R:*>` for the type of `ceStar.field`, and the bad assignment would be allowed.

So we don't do that. Instead, we introduce a new parameter, called a *capture parameter*, in the type of the receiver expression. It stands in for the unknown actual region of an object stored in a variable with partially specified type. The introduction of the new parameter is called *capturing* a type. It occurs in steps 3 and 4(b) of typing field access (§ 4.3.1) and steps 3 and 4(d) of typing method invocation (§ 4.3.3). As discussed in § 3.4.2, capturing RPLs can also occur when inferring arguments to method region parameters.

In this example, the captured type of the receiver expression is `CaptureExample<P>`, where `P` is the capture parameter. `P` is constrained to be included in `R:*`, because that is all we know about the actual region from the type `CaptureExample<R:*>`. This is the only way parameters can be inclusion-constrained in DPJ (§ 2.6.3). In the example above, now the bad assignment is disallowed, because the type `CaptureExample<R:r2>` is not assignable to the type `CaptureExample<P>`, where `P` is the capture parameter. Nor is the type `CaptureExample<R:*>` assignable to `CaptureExample<P>`. However, `CaptureExample<P>` is assignable to `CaptureExample<R:*>`, because of the inclusion constraint.

More generally, this is how the compiler computes the capture of a type in DPJ:

1. Take the normal Java capture of the type, substituting for any wildcard generic type arguments but keeping the same RPL arguments, if any.

2. For each RPL that is partially specified (i.e., contains `*` or `[?]`), replace that RPL with a capture parameter constrained to be included in the RPL.

Notice that if there are no generic wildcards and no partially specified RPL arguments, then the capture operation does nothing to the type.

Capturing types is mostly an internal compiler mechanism. Programmers never have to do it, and they shouldn't even have to worry about it, except to deal with capture errors when they occur. For example, the above code, if compiled, would generate a type error like "expected type `CaptureExample<capture of R:*>`, found type `CaptureExample<R:r2>`." That error will probably be mysterious unless you understand how type capture works.

Here is another common way that capture errors can occur, involving method invocations:

```
1  abstract class CaptureMethodExample<region R> {
2      abstract void callee(CaptureMethodExample<R> x);
3      void caller(CaptureMethodExample<*> y) {
4          // Compile error!
5          y.callee(y);
```

```
6       }
7   }
```

Line 4 causes an error, because it is attempting to assign `y`, which has type

```
CaptureMethodExample<*>
```

to type

```
CaptureMethodExample<capture of *>
```

which is the type of the formal parameter `x` of `callee`, after capturing the type of the receiver expression.

Usually you can work around these errors by adding a parameter. For example, the code above could be rewritten as follows:

```
1  abstract class CaptureMethodExample<region R> {
2      abstract void callee(CaptureMethodExample<R> x);
3      <region R>void caller(CaptureMethodExample<R> y) {
4          // OK
5          y.callee(y);
6      }
7  }
```

By adding a method region parameter `R`, we "capture the type ourselves." Now the code explicitly says that the region in the type of `y` is the same as the region of the type of `x`: whatever it is, it is bound to the same parameter `R` in both cases.

# 5 Effects

Effects are the way that DPJ tracks accesses to the heap to enforce determinism. An effect is an action that reads or writes memory. Every statement and expression in the program is assigned an effect. If the effects of two statements do not interfere, then the statements may be safely run in parallel. Effects don't interfere if neither one writes to memory, or they operate on different parts of memory, or they are both invocations of a method declared `commutative` (§ 3.3.4).

This section describes how effects work in DPJ. § 5.1 describes *basic effects*, which are individual actions involving memory (such as reading or writing a variable). § 5.2 describes *effect summaries*, which are the program representation of sets of basic effects, and are used to summarize the effects of methods. § 5.3 describes *local effects*, which are effects inside a method that are not visible to the caller and so may be omitted from the method's effect summary. § 5.4 explains how DPJ statements and expressions generate basic effects. § 5.5 describes *effect coarsening*, which is necessary for summarizing effects on RPLs and local variables that can go out of scope. § 5.6 explains the *subeffect* relation on effects, which is important for checking method effect summaries against the actual method effects. § 5.7 explains the *noninterference* relation on effects, which is important for checking that pairs of parallel tasks have no conflicting operations.

## 5.1 Basic Effects

A basic effect is an action involving memory. DPJ's effect system represents the following kinds of effects:

1. *Read effects:* A read effect indicates a read operation on an RPL (§ 2) or local variable. Such an effect summarizes one or more reads to one or more memory locations associated with the region or regions named by the RPL.

2. *Write effects:* A write effect indicates a write operation on an RPL (§ 2) or local variable. Such an effect summarizes one or more writes *or reads* to one or more memory locations associated with the region or regions named by the RPL.

3. *Invocation effects:* An invocation effect indicates an invocation of some method, causing some set of basic effects (the effects of invoking the method). The basic effects associated with the invocation are called the *underlying effects* of the invocation. For example, if method `m` has effect summary `writes r` (§ 5.2), then invoking `m` generates an invocation effect with `m` as its method and `writes r` as its underlying effect.

Read and write effects on an RPL $R$ are generated by directly accessing a class field declared to be in $R$ (§ 3.2) or an array cell in $R$ (§ 4.2), or by invoking a method with the effect in its effect summary (§ 3.3.1). Invocation effects are always generated by invoking methods. § 5.4 gives more details on how DPJ statements and expressions generate these effects.

Invocation effects are necessary because some method invocations can commute with others (§ 3.3.4). To keep track of these pairs of commuting methods, the compiler needs to record the information about which method was invoked, in addition to what effects the method invocation caused.

Notice that effects on local variables (including method parameters) are recorded separately from effects on RPLs. Because local variables cannot have their references taken, and never alias, the compiler automatically keeps track of interfering effects on local variables. The programmer doesn't have to put them in RPLs or summarize their effects.

Finally, notice that a write effect can represent both reads and writes. Writes are "stronger" than reads (for interference, at least one of two operations to the same location must be a write), so it is sound but conservative to represent reads as writes. In some cases it may reduce the size of the effect set. For example, a read and a write to the same location may be represented with just the write effect. Finally, allowing write effects to represent reads does not sacrifice any precision, since (1) a read to a location alone can always be represented as a read effect; and (2) a read and write to a location can always be represented as a single write to that location without any loss of precision (the presence of the write already causes any parallel access to the location to interfere, so the presence or absence of the read makes no difference).

## 5.2 Effect Summaries

An *effect summary* is a bit of program text that summarizes a set of basic effects (§ 5.1). In the current DPJ language, effect summaries can appear in the program text only in method definitions, where they summarize the effects of invoking the method (§ 3.3.1). It is anticipated that future versions of DPJ will also allow effect summaries to appear as arguments to *effect variables* in class types and method invocations, for greater flexibility in specifying and checking effects.

An effect summary consists of one of the following:

1. `pure`, indicating no effect on the heap.
2. `reads` *rpl-list*, indicating reads to the RPLs given in *rpl-list*.
3. `writes` *rpl-list*, indicating writes *or reads* to the RPLs given in *rpl-list*.
4. `reads` *rpl-list-1* `writes` *rpl-list-2*, indicating both reads to the RPLs in *rpl-list-1* and writes to the RPLs in *rpl-list-2*.

Currently there is no way to represent an invocation effect (§ 5.1) directly in an effect summary; invocations in the method body must be summarized by giving their underlying effects. This may change in future versions of DPJ.

Here is a simple example of an effect summary:

```
1 class Point<region R> {
2     int x in R, y in R;
3     <region Rp>void add(Point<region Rp> p)
4       reads Rp writes R {
5         this.x += p.x;
6         this.y += p.y;
7     }
8 }
```

This class defines a simple `Point` object with integer coordinates `x` and `y`. There is one class region parameter `R` (§ 3.4.1), and the coordinate fields are placed in the region of `R`. The `add` method takes a `Point` object with some other (possibly different) region `Rp` and adds the coordinates of that point to the coordinates of this one. The summarized effects are `reads Rp writes R`, shown in line 4, because the method reads the coordinates of `p` in region `Rp` and writes the coordinates of `this` in region `R`.

## 5.3 Local Effects

A basic effect (§ 5.1) is *local* if it is a read or write effect on a local variable or local RPL (§ 2.5), or it is an invocation effect whose underlying effects are all local. Local effects are contained within a method scope and never seen by the calling context, so they may be ignored when summarizing method effects. For example:

```
class LocalEffects<region R> {
    int x in R;
    // method has no effects visible to the caller
    void method() pure {
        region r;
        // Write effect on var is local
        LocalEffects<r> var = new LocalEffects<r>();
        // Write effect on region r is local
        var.x = 5;
    }
}
```

## 5.4 Effects of Statements and Expressions

At the heart of DPJ's determinism checking is an analysis of the effect of every statement and expression in the program. To compute the effects, the compiler uses the following information:

- The form of the expression or statement. For example, an assignment statements generates a write on the left-hand side, and a read on the right-hand side.
- The type of a field access receiver, method invocation receiver, or array, together with the RPL of the field (for field access) or declared method efects (for method invocations). For example, if a class `C<region R>` has a field `x in R`, then access through a variable of type `C<r>` generates an effect on `r`. Similarly, if `C` has a method `void m() writes R`, then invoking `m` on a variable of type `C<r>` generates the effect `writes r`.

The field region specifiers (§ 3.2) are important because they effectively partition the class fields into regions that can be used to describe the effects. The method effect summaries (§ 3.3.1) are important because they (1) document the effects of methods at API boundaries (including, e.g., methods in classes for which the source code is not available); and (2) allow the compiler to infer the effect of a method invocation from the summary, rather than doing an interprocedural analysis. (Because method calls can be recursive, this analysis would need to iterate to a fixed point.)

In more detail, here is how the compiler computes the effects of a DPJ statement or expression:

**Field access:** For field access expressions *receiver-exp*`.`*field-name* that directly access a non-`final` field, the compiler first computes the RPL accessed by the expression. It uses the same procedure as for typing field access (§ 4.3.1), except that it uses the RPL specifier of the field (§ 3.2) instead of the type associated with the field, and there are no substitutions for type parameters in step 4(a). The compiler records a write or read effect (§ 5.1) to the RPL so computed, depending on whether the expression appears on the left-hand side of an assignment, or in a read access.

Reads of fields declared `final` generate no effect (writes, other than initialization, are not allowed). Because the value of a `final` field does not change after initialization, reading it cannot cause a conflicting access.

**Array access:** For array access expressions $e_1$`[`$e_2$`]` that access an array cell, the compiler first computes the RPL of the cell. To do this it substitutes $e_2$ for the leftmost index variable in the leftmost RPL argument

appearing in the type of $e_1$. For example, if array `A` has type `int[]<[_]>`, then the RPL accessed by `A[0]` is `[i]`; and if array `B` has type `int[]<[i]>#i[]<[i]:[j]>#j`, then the RPL accessed by `B[0]` is `[0]`. The compiler records a write or read effect to the RPL so computed, depending on whether the array access expression appears on the left-hand side of an assignment, or in a read access.

Note that access through multiple dimensions of an array of arrays causes a read effect for all but the last dimension. For example, for array `B` defined above, the effect of expression `B[0][1]` is a read of `[0]`, plus a either a read or write on `[0]:[1]`, depending on the context. For example, `x = B[0][1]` generates a read, while `B[0][1] = x` generates a write. In fact, this is just a special case of the rules stated in the previous paragraph, together with the rules for compound expressions (see below).

**Method invocation:** For method invocation expressions, the compiler computes and accumulates the effects of evaluating the receiver and argument expressions as described in this section. Then it uses the method's effect summary (§ 3.3.1) to compute the effect of the invocation itself. It uses the same procedure as for computng the return type of a method (§ 4.3.3), except that it uses the effect summary instead of the declared return type, and there are no substitutions for type parameters in steps 4(a) or 4(b). The compiler records an invocation effect (§ 5.1) with the invoked method and the computed effect.

**Local variable access:** For statements and expressions that access a non-`final` local variable (i.e., variable declared in a method scope or method formal parameter), the compiler records a read or write effect on the variable. Effects on `final` local variables are ignored.

**Compound statements and expressions:** For any statement or expression made up of other statements or expressions (including, for example, an assignment that has a method invocation on its right-hand side, or an array access to an expression which is itself an array access), the compiler accumulates the effect of the components, coarsening component effects as necessary (§ 5.5) to remove elements that are no longer in scope at the outer level.

## 5.5 Effect Coarsening

A statement or expression may generate an effect that is no longer in scope where the effect must be reported. For example, consider the following `foreach` loop (§ 6.2) on an index-parameterized array (§ 3.5.3):

```
1 int[]<[_]> A = new int[N]<[_]>;
2 foreach (int i in 0, N) {
3     A[i] = i;
4 }
```

In line 3, the effect is `writes [i]`. But what is the effect in the scope outside the loop? This is important, for example, if this code appears inside a method body and its effect must be summarized. We can solve this problem with partially-specified RPLs (§ 2.4). For example, the effect `writes [?]` covers all the effects `writes [i]` for all `i` inside the loop, so that is what we use. This is called *effect coarsening*.

**Coarsening of local effects:** An effect on a local RPL (§ 2.5) or local variable is simply deleted from the effect set when the RPL or variable goes out of scope. For example, in the following code fragment, neither block contained in the `cobegin` has any effect:

```
class LocalEffectCoarsening<region R> {
    int x in R;
    void method() pure {
        cobegin {
            // No effect here because var1 and r1 are out of scope
            {
                region r1;
                LocalEffectCoarsening<r1> var1 =
                    new LocalEffectCoarsening<r1>();
                var1.x = 10;
            }
            // No effect here either because var2 and r2 are out of
            // scope
```

```
            {
                region r2;
                LocalEffectCoarsening<r2> var2 =
                    new LocalEffectCoarsening<r2>();
                var2.x = 25;
            }
        }
    }
}
```

This pattern can also be used effectively with `foreach`, by having each iteration create its own objects whose effects are invisible to the other iterations:

```
foreach (int i in 0, N) {
    // r is local to a foreach iteration
    region r;
    // Do some effects on r
    ...
}
```

This technique is useful for creating private objects in each iteration that manipulate data local to that iteration. See *The Deterministic Parallel Java Tutorial* for more examples.

**Coarsening of nonlocal effects:** Coarsening of nonlocal effects works as follows.

*Variable RPLs:* An effect on an RPL starting with a `final` local variable (§§ 2.1.5, 2.2) is coarsened when the variable goes out of scope. The effect is replaced by a new effect on *owner-rpl*`:*`, where *owner-rpl* is the owner RPL of the variable's type (§ 4.1.4). Because variables may be nested under variables, this operation is performed recursively on the resulting RPL until an RPL is obtained that is valid in the outer scope.

For example:

```
class VariableCoarsening<region R> {
    int x in R;
    void method() writes R:* {
        // Coarsened effect is 'writes R:*'
        {
            final VariableCoarsening<R> vc1 =
                new VariableCoarsening<R>();
            VariableCoarsening<vc1> vc2 =
                new VariableCoarsening<vc1>();
            // Effect is 'writes vc1'
            vc2.x = 5;
    }
}
```

Note that method parameters are in scope in the method definition, so effects on method parameter RPLs don't need to be coarsened in the method's effects:

```
abstract class ParamRPLs<region R> {
    abstract void method(ParamRPLs<R> param)
        // OK; 'writes R:*' is also OK, but less precise
        writes param;
}
```

*Array index RPLs:* An effect on an RPL containing the array index `[`*e*`]` is coarsened to `[?]` if the expression *e* includes an integer variable that goes out of scope. An example is the `foreach` code given at the beginning of this section.

## 5.6 Subeffects

The subeffect relation determines whether one set of effects covers another set of effects, i.e., all effects in the second set are represented in the first set. The compiler uses the subeffect relation to check that a method's declared effects (§ 3.3.1) include the actual effects of its body and the effects of any overriding methods.
**Basic effects:** The following rules determine when one basic effect (§ 5.1) is a subeffect of another:

- If one RPL is included in another (§ 2.6.3), then a read of the first RPL is a subeffect of a read or write to the second. For example, `reads Root` is a subeffect of `reads Root:*` and `writes Root:*`.
- If one RPL is included in another, then a write to the first RPL is a subeffect of a write to the second. For example, `writes Root` is a subeffect of `writes Root:*`. Note that `writes Root` is *not* a subeffect of `reads Root:*`.
- An invocation of method $m$ with underlying effect set $E_1$ is a subeffect of an invocation of method $m$ with underlying effect set $E_2$ if $E_1$ is a subeffect of $E_2$. For example, an invocation of $m$ with underlying effect `writes R` is a subeffect of an invocation of $m$ with underlying effect `writes R:*`.

**Effect sets:** The following rules determine when one set of basic effects is a subeffect of another:

- If each basic effect in one set is a subeffect of some basic effect in the another set, then the first set is a subeffect of the second. For example, `reads A writes B` is a subeffect `writes A:*, B:*`, because `reads A` is a subeffect of `writes A:*`, and `writes B` is a subeffect of `writes B:*`.
- An invocation effect is a subeffect of its underlying effect. For example, an invocation of method $m$ with the effect `writes R` is a subeffect of `writes R`. This means that invocations may always be summarized by stating their underlying effects.

## 5.7 Noninterference of effect

The noninterference relation on effects determines whether two sets of effects are safe to be run in parallel. The compiler uses the noninterference relation to check that there are no conflicting effects in mutually parallel tasks (§ 6).
**Basic effects:** The following rules determine when one basic effect (§ 5.1) is a subeffect of another:

- Two basic effects are noninterfering if they are both read effects.
- Two basic effects are noninterfering if they are each read or write effects, and they operate on disjoint RPLs (§ 2.6.4).
- Two invoke effects are noninterfering if they invoke the same method, and the method is declared `commutative` (§ 3.3.4).

**Effect sets:** The following rules determine when one effect set is a subeffect of another:

- If every basic effect in one set is noninterfering with every basic effect in another set, then the two sets are noninterfering. For example, `reads A writes B` is noninterfering with `writes C,D`. However, `reads A writes B` is interfering with `writes A,C`, because `reads A` inteferes with `writes A`.
- If two effect sets are each a subeffect of another effect set, and the including sets are noninterfering, then the included sets are noninterfering. In particular, two invoke effects are noninterfering if their underlying effects are (because the underlying effects include the invoke effects, see § 5.6).

# 6 Parallel Control Flow

DPJ employs a fork-join model of parallelism. That means that a task may launch several parallel tasks (the fork) and all must complete (the join) before the launching task can continue. Recursive forking is supported, i.e., tasks can launch other tasks to arbitrary depth. There are two ways to fork tasks: `cobegin`, which forks several statements as parallel tasks, and `foreach`, which forks groups of consecutive loop iterations as parallel tasks.

## 6.1 `cobegin`

The syntax of the `cobegin` statement is as follows, where $S$ is a DPJ statement:

`cobegin` $S$

If $S$ is any statement but a block enclosed in curly braces `{...}`, or if $S$ is a block consisting of a single statement, then the `cobegin` just executes the statement $S$. Otherwise, the component statements of $S$ are run as parallel tasks. There is an implicit barrier (join) at the end of the `cobegin` statement, so that all the component tasks must finish execution before the parent task executes the statement after the `cobegin`.

For example, the following code executes statements `S1` and `S2` in parallel:

```
cobegin {
  S1;
  S2;
}
S3;
```

Because `S3` appears after the `cobegin`, both `S1` and `S2` are guaranteed to finish before `S3` is executed.

In order to guarantee deterministic execution, the compiler checks the component statements of a `cobegin` for noninterference (Section 5.7). If interference is discovered, then the compiler issues a warning. For example, the following code would cause a warning, because of the interfering writes to variable `x` in the parallel tasks:

```
class C {
  void m() {
    int x;
    cobegin {
      x = 0;
      x = 1;
    }
  }
}
```

On the other hand, the following code would compile with no warning:

```
class C {
  void m() {
    int x, y;
    cobegin {
      x = 0;
      y = 1;
    }
  }
}
```

The `cobegin` statement is most often used with recursion. The following pattern is typical:

```
void recursiveMethod(...) {
  if (...) {
```

```
    // do base case sequentially
  } else {
    cobegin {
      recursiveMethod(...);
      recursiveMethod(...);
    }
  }
}
```

Note that the "recursion cutoff" (i.e., when the base case takes over) has to be programmed manually. For example, in a parallel recursive sort, the condition might say to do the sort sequentially when the input array reaches a certain minimum size. The minimum should be chosen so that (1) there are enough parallel tasks for the scheduler to balance the computation, but (2) the task creation overhead is not unduly large. See *The Deterministic Parallel Java Tutorial* for more examples of `cobegin`.

## 6.2 `foreach`

`foreach` operates similarly to `cobegin`, except that the parallel tasks are the iterations of a loop, instead of the component statements of a block. The granularity of parallelism (i.e., how many loop iterations to execute in a task) is controllable by the programmer.

### 6.2.1 Writing the `foreach` loop

The syntax of the `foreach` loop is as follows:

`foreach (int` *index-var* `in` *start*`,` *length*`,` *stride*`)` *body*`;`

`index-var` is an identifier, *start*, *length*, and *stride* are integer expressions, and *body* is a statement. The *stride* expression is optional, and the default is 1. If it appears, the *stride* expression must evaluate to an integer greater that 0. The `foreach` loop executes the loop body once for each element of an iteration space given by all integers $stride \cdot i$ such that $i$ ranges between 0 and $length - 1$, inclusive. The variable *index-var* may not be modified in the loop body.

For example, the following code sets the cells of array `A` with even indices to 0:

```
foreach (int i in 0, A.length, 2) {
  A[i] = 0;
}
```

The compiler performs the following noninterference check for indexed `foreach` loops:

1. Infer the effect set of *body* (Section 5.4).
2. Create a copy of the effect set generated in (1), but replace every occurrence of *index-var* with a fresh variable that is known to be unequal to *index-var*. This simulates the effects generated by two distinct iterations of the loop, for which *index-var* will have distinct values.
3. Check whether the effect sets generated in (1) and (2) are noninterfering (Section 5.7).

If interference is detected, the compiler issues a warning.

The DPJ runtime divides the `foreach` iterations into parallel work according to the programmer-specified granularity (Section 6.2.2). As in the case of `cobegin`, there is an implicit barrier at the end of the `foreach`; i.e., all tasks created by the `foreach` must complete before any code following the `foreach` is executed.

### 6.2.2 Controlling the granularity of parallelism

In most cases, it would be inefficient to issue every loop iteration as a parallel task. For example, consider a loop iterating over an array with 100,000 elements on a machine with 10 cores. If 100,000 tasks were issued to do this computation, the task creation overhead would swamp the parallelism.

Instead, the DPJ compiler allows the user to control the granularity of task creation. The user can specify a minimum task size, in terms of the number of iterations. The compiler recursively splits the iteration space, until the minimum size is reached. For example, if the cutoff were specified to be 1000 in the example above, then the compiler would divide the loop into 100 tasks of 1000 iterations each.

If the `foreach` computation is perfecly balanced (i.e., each iteration does exactly the same amount of work), then it makes sense to make the cutoff $I/T$, where $I$ is the number of loop iterations and $T$ is the number of threads available to the DPJ runtime (usually equal to the number of cores on the host machine). Using this strategy, in the example with 100,000 iterations and 10 cores, the cutoff should be 10,000.

If the computation is not perfectly balanced, then a better strategy is to *over-decompose* the computation, specifying more tasks than the number of available threads. This allows the scheduler to schedule multiple tasks of varying size per thread, in a such a way that the aggregate amount of work per thread is roughly balanced. As in the case of the `cobegin` cutoff (Section 6.1), the programmer should choose a minimum task size that creates enough parallel work without incurring too much task creation overhead.

The precise mechanism for specifying the number of DPJ threads and the `foreach` cutoff is implementation-dependent. *The Deterministic Parallel Java Installation Manual* explains how this mechanism works for the `dpjc` compiler.

# 7 The DPJ Runtime

This section gives an overview of the classes `ArraySlice` and `Partition` in the package `DPJRuntime`. These classes are part of the runtime supplied with DPJ and are useful for manipulating arrays. For full documentation of these classes, and for coverage of the other DPJ runtime classes, see the HTML documentation included in the DPJ release at

`DPJ/Implementation/Runtime/dpjdoc`.

To use the DPJ runtime, you have to do the following:

1. Put `import DPJRuntime.*` at the top of your code (or just import the class or classes you want to use), as for regular Java.

2. When compiling your code, you must compile the runtime classes located at `Implementation/Runtime/dpj` in the DPJ release together with any code that uses the runtime. For example, if class file `Foo.java` depends on the DPJ runtime, then you need to do something like this:

   ```
   dpj Foo.java ${DPJ_ROOT}/Implementation/Runtime/dpj/*java
   ```

   *It is not sufficient to put the runtime classes in your class path.* If you do that, you will get a lot of errors. The reason is that DPJ bytecode doesn't yet properly support separate compilation of DPJ annotations; the DPJ compiler needs the DPJ source for all annotated code to process the annotations.

3. When running your code, put the runtime classes located at `Implementation/Runtime/classes` in your class path, as for regular Java.

## 7.1 ArraySlice

In parallel algorithms that operate on arrays, especially divide-and-conquer algorithms, it is often necessary to split an array into parts and operate separately on the separate parts. DPJ provides array slices as an operation of a library class, called `ArraySlice`. There are two kinds of `ArraySlice` classes: the generic `ArraySlice` class, and a set of `ArraySlice` classes specialized to primitive types.

**Generic ArraySlice:** The generic `ArraySlice` class represents an array of objects. It has one type parameter and one RPL parameter:

```
class ArraySlice<type T, region R>
```

The type parameter is the element type of the array, and the RPL parameter is the RPL of the array storage.
*Creating an ArraySlice:* There are three ways to create a `ArraySlice` object. The first is to call a constructor that takes just a length argument. This operation creates a new `ArraySlice` with the specified length, type, and RPL. For example, the following code creates an `ArraySlice` storing 10 `Integer` objects, and the storage is in RPL `r`:

```
region r;
ArraySlice<Integer,r> A = new ArraySlice<Integer,r>(10);
```

Creating a fresh DPJ array is useful, but sometimes you have a Java array and you want to make an `ArraySlice` out of it. So the second way to create an `ArraySlice` is to call a constructor that takes an ordinary Java array as an argument. For example, the following code creates a Java array of 10 `Integer` objects, and wraps it in a `ArraySlice`:

```
region r;
Integer[]<r> a = new Integer[10]<r>;
ArraySlice<Integer,r> A = new ArraySlice<Integer,r>(a);
```

Note that the type and RPL of the array being passed in (here, `Integer` and `r`) must match the type and RPL argument of the `ArraySlice` type; if not, a compile error occurs.

The third way to create an `ArraySlice` is to make a *subslice* (i.e., a slice) of an existing `ArraySlice`. This is explained below.
*Accessing elements:* `ArraySlice` has `put` and `get` operations similar to the ones in `java.util.ArrayList`. For example, if `A1` and `A2` are `ArraySlices`, then the following code fragment gets element 0 out of `A1` and stores it into element 0 of `A2`:

```
A2.put(0, A1.get(0));
```

If the `ArraySlice` was created by wrapping a Java array, then the `put` operation modifies the wrapped array. The effect of a `get` operation is a read on the region of the `ArraySlice`, and the effect of a `put` operation is a write to the region of the `ArraySlice`.

Accesses are bounds-checked. Any attempt to access a position less than 0 or greater than the array length minus one throws an `ArrayIndexOutOfBoundsException`.
*Subslices:* The real usefulness of `ArraySlice` is its support for *subslices*, which are contiguous subsections of an array. To create a subslice of an `ArraySlice`, you call the `subslice` instance method with a start and length argument. For example, the following code creates an `ArraySlice` and then extracts the subslice of length 2 starting at position 5 (here we use the default RPL of `Root`):

```
ArraySlice<Integer> A = new ArraySlice<Integer>(10);
ArraySlice<Integer> B = A.subslice(5,2);
```

There are two nice things about subslices. First, creating a subslice takes minimal time and space overhead. Nothing is copied, and no new storage allocated to hold any array elements. A small object is created that stores the start position, length, and a reference to the underlying array.

Second, as far as its API is concerned, a subslice is indistinguishable from a freshly created `ArraySlice`. For example, the subslice created above is zero-indexed, it has length 2, and attempts to access indices other than 0 and 1 throw an exception. However, the subslice also provides a view into the original array. For example, the following code stores 1 into position 0 of `B`, which is the same as position 5 of `A`:

```
B.put(0,1);
```

You can get the start position out of an `ArraySlice` by reading the field `start`, and you can get the length by reading the field `length`.

These features allow methods that operate on array subranges to be zero-indexed, without worrying about the index parameters of the subranges. For example, here is some simple recursive code that increments every position of an `ArraySlice` by 1:

```
<type T, region R>void increment(ArraySlice<T,R> A) {
    if (A.length == 0) return;
    if (A.length == 1) {
        A.put(0, A.get(0)+1);
    }
    int mid = A.length / 2;
    increment(A.subslice(0, mid));
    increment(A.subslice(mid, length-mid));
}
```

Without the subslice feature, this code would have to be written by passing the index ranges as parameters to the `increment` method, which is ugly.

**Specialized ArraySlices:** Because Java does not support binding primitive types to generic parameters, the DPJ runtime also has versions of `ArraySlice` specialized to the primitive types (`DPJInt` for `int`, `DPJBoolean` for `boolean`, etc.). These operate identically to the generic `ArraySlice`, except that the element type is given by the class, and there is no generic parameter.

Another way to write such arrays is to create a generic `ArraySlice` using the class corresponding to the primitive type. For example, instead of an `ArraySliceInt`, one could use a `ArraySlice<Integer>`, as described above. This works, but the code is more verbose, as well as more memory-intensive and slower, as Java has to box and unbox all those primitive types when putting them into and getting them out of the array.

## 7.2 Partition

For parallel divide-and-conquer algorithms on arrays, it is often important to create disjoint collections of subslices. For example, a parallel sorting algorithm might repeatedly divide an array into disjoint halves (or quarters, etc.) and operate recursively in parallel on the pieces. To support effect checking for this kind of algorithm, the DPJ runtime includes a class `Partition` for representing disjoint collections of subslices of the same array. Each of the subslices in the collection is called a *segment* of the partition. As with `ArraySlice`, there is a generic version, and there are specialized versions.

**Generic `Partition`:** The generic `Partition` class represents an array of subslices of an `ArraySlice`. It has one type parameter and one RPL parameter:

```
class Partition<type T, region R>
```

The type parameter is the element type of the `ArraySlice` being partitioned, and the RPL parameter is the RPL of the array storage.

*Creating a `Partition`:* There are several ways to create a `Partition`; for a full list, see the HTML documentation. Here are two useful ways. First, `Partition` has a constructor that takes an `ArraySlice` to partition, an index at which to partition, and a `boolean` value that says whether to exclude or include the element at the index position. For example, if `A` is an `ArraySlice<Integer>` of length 10, then

```
new ArraySlice<Integer>(A, 5, true)
```

partitions `A` into the segments [0,4] and [6,9] (excluding position 5), while

```
new ArraySlice<Integer>(A, 5, false)
```

partitions `A` into the segments [0,4] and [5,9] (including position 5). This constructor is useful for parallel divide-and-conquer algorithms with a fanout of 2.

Second, `Partition` has a static factory method `stridedPartition` that takes an `ArraySlice` to partition and a stride at which to partition. For example, if `A` is a `ArraySlice<Integer>` of length 10, then

```
new Partition<Integer>(A, 2)
```

creates a `Partition<Integer>` with five segments, each of length 2. This feature is useful for parallel divide-and-conquer algorithms with a fanout of greater than two, as well as flat partitions (such as tiling an array).
*Accessing segments:* The field `length` stores the number of segments in the partition. It is `final`, so reading it has no effect (§ 5.4). The method `get` takes an integer index `idx` and returns the segment corresponding to that index (and throws an exception if the index is out of range). The type of the segment is `ArraySlice<T,this:[idx]:*>`.

The index-parameterized type returned by `get` allows the different segments to be operated on in parallel without interference, similarly to an index-parameterized array (§ 3.5.3). For example, the following code uses `Partition` to parallelize the simple recursive `increment` shown in § 7.1:

```
<type T, region R>void parallelIncrement(ArraySlice<T,R> A)
  writes R:* {
    if (A.length == 0) return;
    if (A.length == 1) {
        // Effect is 'writes R'
        A.put(0, A.get(0)+1);
    }
    int mid = A.length / 2;
    final Partition<T,R> segs = new Partition<T,R>(A,mid)
    cobegin {
        // Effect is 'writes segs:[0]:*'
        parallelIncrement(segs.get(0));
        // Effect is 'writes segs:[1]:*'
        parallelIncrement(segs.get(1));
    }
}
```

For more examples of how to use `Partition`, see *The Deterministic Parallel Java Tutorial*.

The variable `this` in the type gets substituted by the recevier expression at the call site (§ 4.3.3), so it is usually most useful to call `get` through a `final` local variable. Including the variable in the RPL ensures that the compiler doesn't erroneously infer disjointness for two different partitions of the same array. For example, in the following code, `segs1.get(1)` and `segs2.get(0)` are not disjoint (they overlap at [2,7]):

```
ArraySlice<Integer> A = new ArraySlice<Integer>(10);
// Create segments [0,1] and [2,9]
Partition<Integer> segs1 = new Partition(A, 2);
// Create segments [0,7] and [8,9]
Partition<Integer> segs2 = new Partition(A, 8);
```

**Specialized `Partition`s:** As with `ArraySlice`, there are versions of `Partition` specialized to the various primitive types. They are provided for convenience and efficiency, as Java does not support binding primitive types to generic type parameters.

# 8 Exception Behavior

In DPJ, an exception thrown outside any parallel construct (`cobegin` or `foreach`) behaves exactly as in sequential Java. An exception thrown inside a parallel construct and caught inside that same parallel construct also behaves as in sequential Java. An exception thrown inside a parallel construct and not caught inside that parallel construct has the following behavior:

1. If an exception $E$ is thrown in branch $B$ of a `cobegin` or `foreach`, then branch $B$ behaves as if it were executed in isolation, starting with the state that existed at the start of the `cobegin` or `foreach`. For example, consider this code, where `loopBody(int)` is a method:

   ```
   foreach (int i in 0, 10) {
       loopBody(i);
   }
   ```

If iteration $I$ throws exception $E$, then replacing the entire `foreach` with `loopBody(I)` would also cause $E$ to be thrown, at the same point in the execution of method `loopBody` as in the parallel case.

2. If multiple branches of a `cobegin` or `foreach`, each run in isolation, would throw an exception, then one of those exceptions is guaranteed to be thrown by the entire `cobegin` or `foreach`. Which one is thrown is scheduler dependent (i.e., different ones may be thrown on different executions).

3. Methods annotated `commutative` (Section 3.3.4) must respect this exception behavior. For instance, method invocations $I_1$ and $I_2$ are not considered to commute with each other if execution $I_1;I_2$ throws an exception, but $I_2;I_1$ does not.