DCL58-J. Enable compile-time type checking of variable arity parameter types

A variable arity (aka *varargs*) method is a method that can take a variable number of arguments. The method must contain at least one fixed argument. When processing a variable arity method call, the Java compiler checks the types of all arguments, and all of the variable actual arguments must match the variable formal argument type. However, compile-time type checking is ineffective when `Object` or generic parameter types are used [Bloch 2008]. The presence of initial parameters of specific types is irrelevant; the compiler will remain unable to check `Object` or generic variable parameter types. Enable strong compile-time type checking of variable arity methods by using the most specific type possible for the method parameter.

Noncompliant Code Example (`Object`)

This noncompliant code example sums a set of numbers using a variable arity method that uses `Object` as the variable arity type. Consequently, this method accepts an arbitrary mix of parameters of any object type. Legitimate uses of such declarations are rare (but see the "Applicability" section of this guideline).

```
double sum(Object... args) {
   double result = 0.0;
   for (Object arg : args) {
     if (arg instanceof Byte) {
       result += ((Byte) arg).byteValue();
     } else if (arg instanceof Short) {
       result += ((Short) arg).shortValue();
     } else if (arg instanceof Integer) {
       result += ((Integer) arg).intValue();
     } else if (arg instanceof Long) {
       result += ((Long) arg).longValue();
     } else if (arg instanceof Float) {
       result += ((Float) arg).floatValue();
     } else if (arg instanceof Double) {
       result += ((Double) arg).doubleValue();
     } else {
       throw new ClassCastException();
     }
   }
   return result;
}
```

Compliant Solution (`Number`)

This compliant solution defines the same method but uses the `Number` type. This abstract class is general enough to encompass all numeric types, yet specific enough to exclude nonnumeric types.

```
double sum(Number... args) {
  // ...
}
```

Noncompliant Code Example (Generic Type)

This noncompliant code example declares the same variable arity method using a generic type parameter. It accepts a variable number of parameters that are all of the *same* object type; however, it may be any object type. Again, legitimate uses of such declarations are rare.

```
<T> double sum(T... args) {
  // ...
}
```

Compliant Solution (Generic Type)

This compliant solution defines the same generic method using the `Number` type.

```
<T extends Number> double sum(T... args) {
  // ...
}
```

Be as specific as possible when declaring parameter types; avoid `Object` and imprecise generic types in variable arity methods. Retrofitting old methods containing final array parameters with generically typed variable arity parameters is not always a good idea. For example, given a method that does not accept an argument of a particular type, it could be possible to override the compile-time checking—through the use of generic variable arity parameters—so that the method would compile cleanly rather than correctly, causing a runtime error [Bloch 2008].

Also, note that autoboxing prevents strong compile-time type checking of primitive types and their corresponding wrapper classes. For instance, this compliant solution produces the following warning but works as expected:

```
Java.java:10: warning: [unchecked] Possible heap pollution from parameterized vararg type T
 <T extends Number> double sum(T... args) {
```

Applicability

Injudicious use of variable arity parameter types prevents strong compile-time type checking, creates ambiguity, and diminishes code readability.

Variable arity signatures using `Object` and imprecise generic types are acceptable when the body of the method lacks both casts and autoboxing and it also compiles without error. Consider the following example, which operates correctly for all object types and type-checks successfully:

```
<T> Collection<T> assembleCollection(T... args) {
  return new HashSet<T>(Arrays.asList( args));
}
```

In some circumstances, it is necessary to use a variable arity parameter of type `Object`. A good example is the method `java.util.Formatter.format(String format, Object... args)`, which can format objects of any type.

Automated detection is straightforward.

Bibliography

[Bloch 2008]	Item 42, "Use Varargs Judiciously"
[Steinberg 2008]	"Using the Varargs Language Feature"
[Oracle 2011b]	Varargs