

Tips and Tricks for Using Language Features in API Design and Implementation

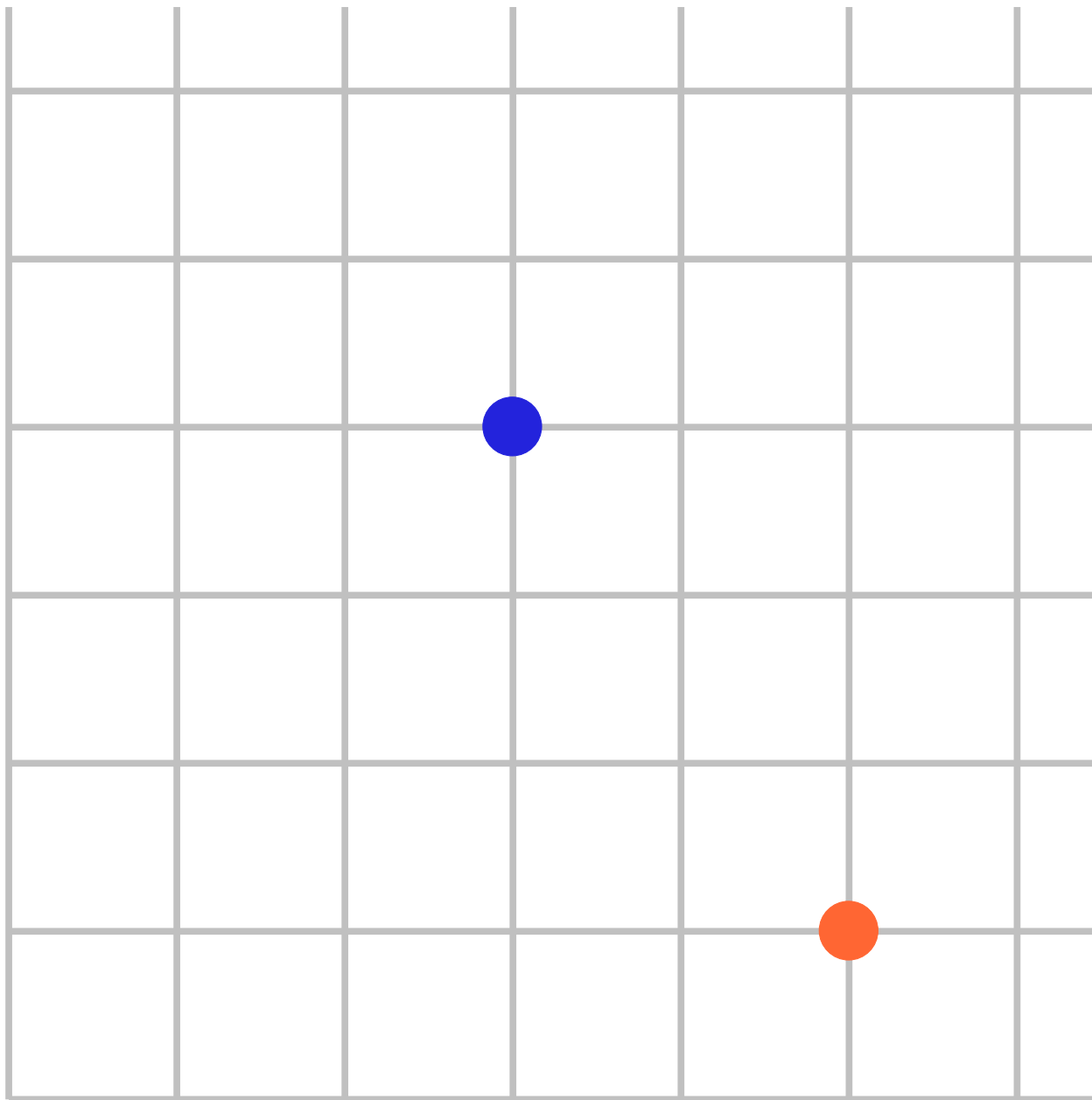
Joseph D. Darcy
joe.darcy@sun.com
<http://blogs.sun.com/darcy/>
Sun Microsystems, Inc.

Outline

- Digression: norms
- Compatibility
- Having your cake and eating it too
- Generics and the Mandelbrot set
- Miscellaneous
- Q&A

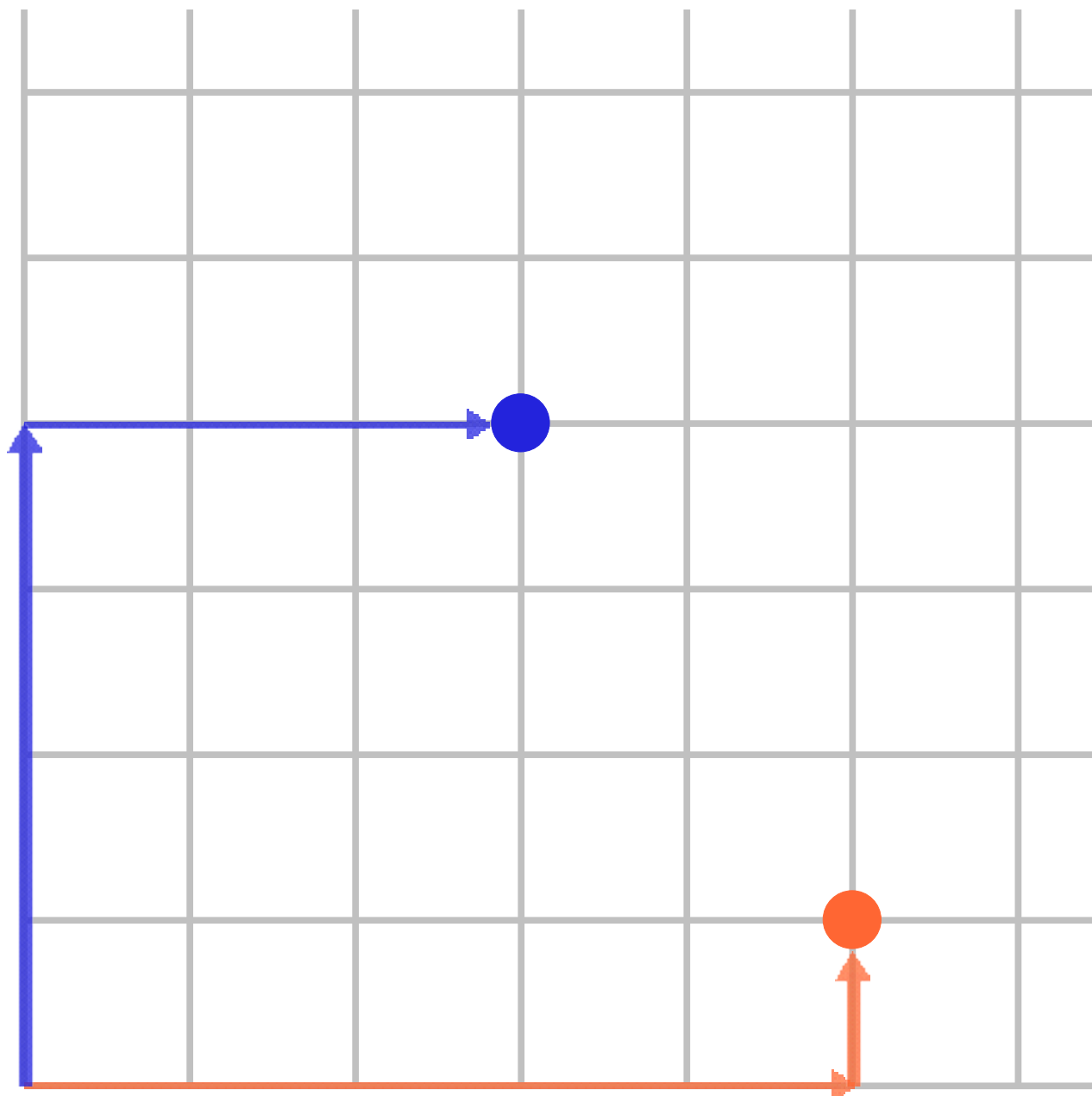
Digression: norms

- How to sensibly measure the size of a collection of numbers?
 - Vectors
 - Benchmark scores
- Many valid answers



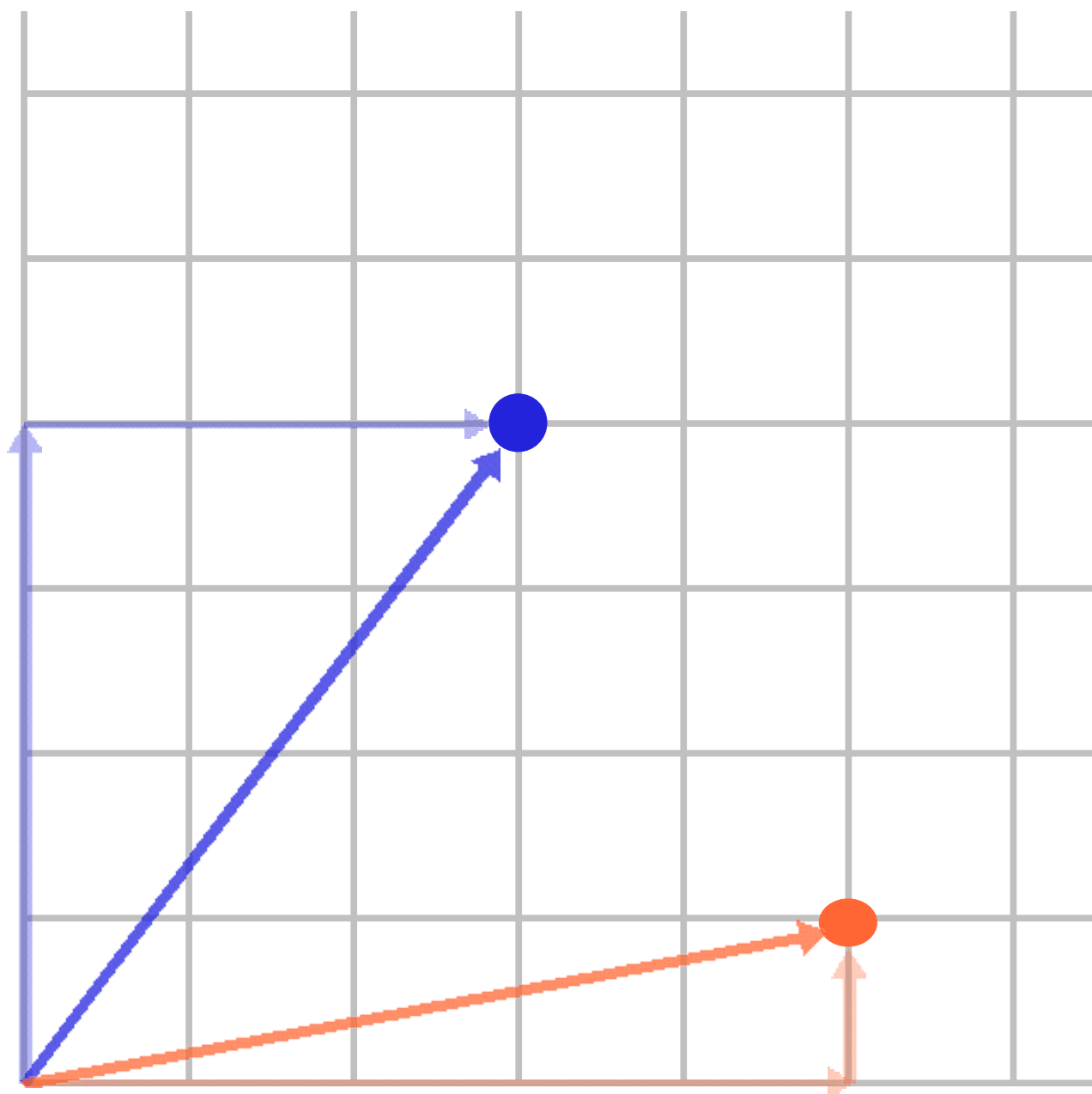
(3, 4)

(5, 1)



$$N_1((3, 4))=7$$

$$N_1((5, 1))=6$$



$$N_1((3, 4))=7$$

$$N_2((3, 4))=5$$

$$N_1((5, 1))=6$$

$$N_2((5, 1))\approx 5.1$$

So what?

- All norms are equivalent:
if one norm is getting close to zero,
all of them are
- *However*, norms can give different answers on the
relative size of input
- To make fine distinctions and good decisions
must know *what* is important and *why*

Importance of Compatibility

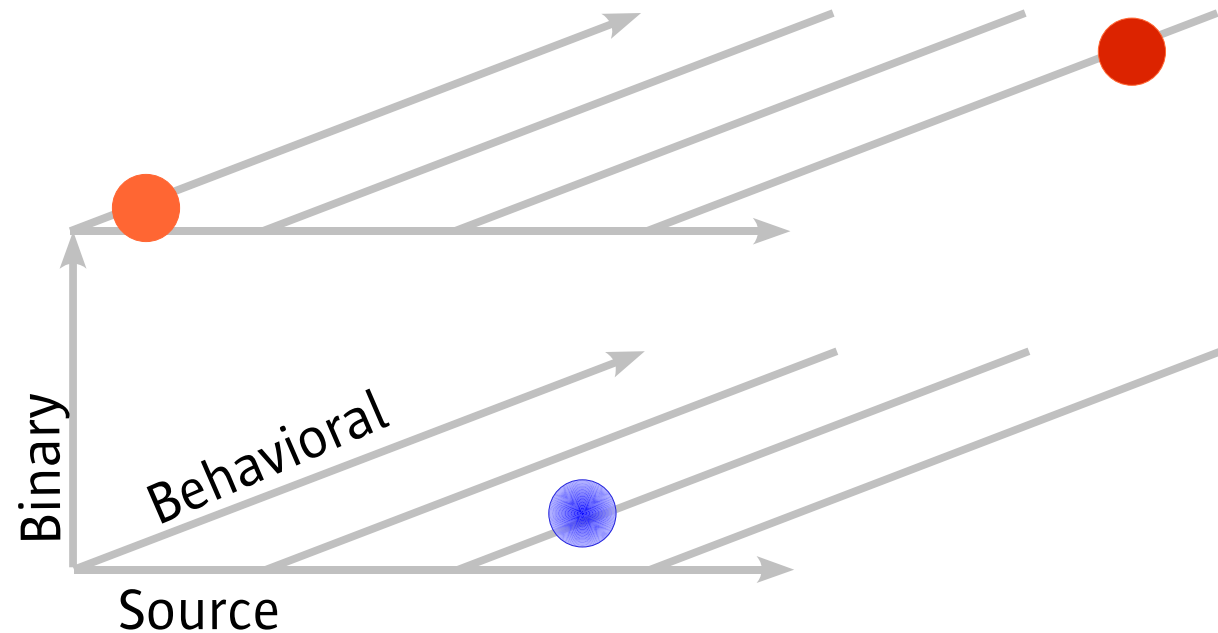
Jake Gittes: *Why are you doing it?
How much better can you eat?
What can you buy
that you can't already afford?*

Noah Cross: *The future, Mr. Gitts,
the future.
–Chinatown*

- (Greatly) constrains future API evolution
- Balance time to market versus feature set
- Provide enough functionality for today and provide enough growth potential for tomorrow

Kinds of Compatibility

- Binary
- Source
- Behavioral



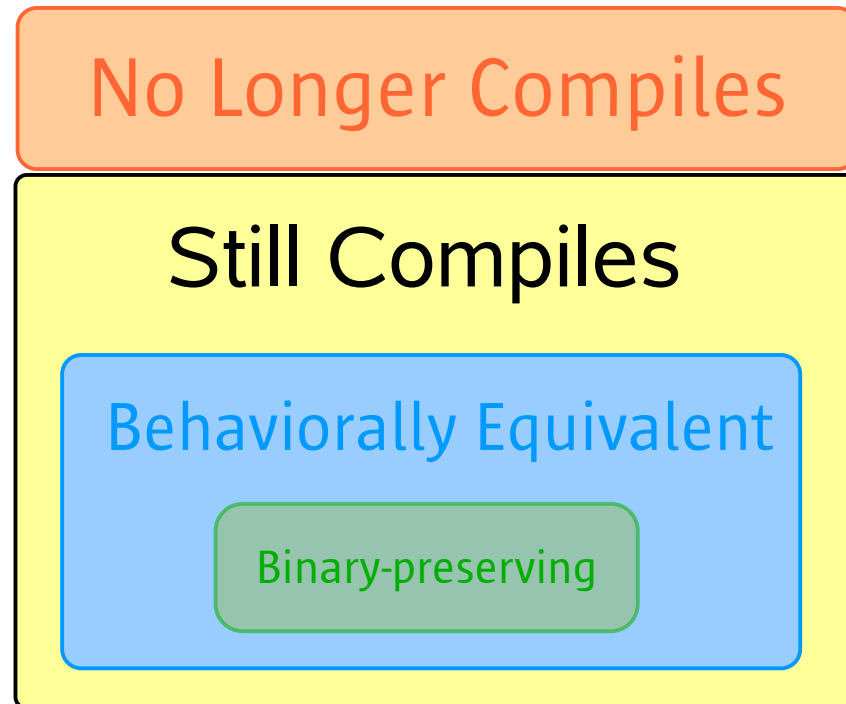
Binary Compatibility

- Defined in JLSv3 Chapter 13
 - “When I use a word,” Humpty Dumpty said in rather a scornful tone, “it means just what I choose it to mean — neither more nor less.”
“The question is,” said Alice, “whether you *can* make words mean so many different things.”
“The question is,” said Humpty Dumpty, “which is to be master — that's all.”
— Lewis Carroll, *Through the Looking Glass*
- Does the program still link?
That's it!
- True/False property

Source Compatibility

- What does this mean?
- Concerns mapping of names used in source code to *binary names* used in class files
 - Can stop compilations after most changes with malicious `import` statements
 - Only consider source using *fully qualified names*
- *Not* just a True/False property

Source Compatibility Threat Levels



```
public final class C {  
    double foo(double d) {  
        return d * 2.0;  
    }  
}
```

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

```
public final class C {  
    double foo(double d) {  
        return d * 2.0;  
    }  
}
```

Delete foo

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

```
public final class C {  
    double foo(double d) {  
        return d * 2.0;  
    }  
}
```

Delete foo

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

Add
`int bar() {
 return 42;
}`

```
public final class C {  
    double foo(double d) {  
        return d * 2.0;  
    }  
}
```

Delete foo

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

Add
int bar() {
 return 42;
}

Add
void double foo(int i) {
 return i * 2.0;
}

```
public final class C {  
    double foo(double d) {  
        return d * 2.0;  
    }  
}
```

Delete foo

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

Add
int bar() {
 return 42;
}

Add
void double
foo(long e1) {
 return (double)
 (e1 * 2L);
}

Add
void double foo(int i) {
 return i * 2.0;
}

Clients

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

Extenders & Implementors

No Longer Compiles

Still Compiles

Behaviorally Equivalent

Binary-preserving

Behavioral Compatibility

- What does it *do*?
- Is it “the same”?
 - Hard to define
 - Depends on what you're looking at
- What about
 - Timing?
 - Reflection?
 - Stack traces?

Compatible?

```
// Original Implementation
enum MetaSyntaticVariable {
    FOO(21),

    BAR(42);

    public int answer() {
        return this.answer;}

    private int answer;
    MetaSyntaticVariable(int answer){
        this.answer=answer;}
}
```

```
// New Implementation
enum MetaSyntaticVariable {
    FOO {
        public int answer() {
            return 21;}}},

    BAR {
        public int answer() {
            return 42;}}};

    public abstract int answer();
}
```

Compatibility Suggestions

- Understand constraints of your users
 - Are all the subtypes known?
 - Are all the clients known?
 - What is being relied on?
- What kinds of changes do you anticipate?
- What is an acceptable threat?

Warmup: Var-args

Consider

```
public class C {  
    public static void main(String... args){  
        System.out.println("Hello World.");  
    }  
}
```

Using Var-args

- When usually create a new array (printf)
- Last parameter is primarily used for grouping *not* conceptually as an array

- Examples:

Yes:

```
j.l.reflect.Array.newInstance(Class<?> compType,  
    int... dimensions);  
j.l.r.Constructor(Object... initargs);  
j.u.Arrays.asList(T... a);
```

No:

```
j.u.Arrays.equals(Object[] a1, Object[] a2);  
j.u.Arrays.deepEquals(Object[] a1, Object[] a2);
```

How to find var-args candidates

- Use an *annotation processor*
- As of JDK 6, standardized meta-programming plugins for **javac**
javac -processor MyProc files
- Can run during a build
- Can process source files or class files
- Can emit user-defined notes/errors/warnings based on analysis of program structure
 - See sample naming convention checker:
sample/javac/processing/src/CheckNameProcessor.java

```

@SupportedAnnotationTypes("*") // Process (examine) everything
@SupportedSourceVersion(RELEASE_6) // Process (examine) everything
public class VarArgProc extends AbstractProcessor {
    private Messenger messenger;
    private ElementScanner6<Void, Void> varArgChecker;

    @Override
    public boolean process(Set<? extends TypeElement> annotations,
                          RoundEnvironment roundEnv) {
        if (!roundEnv.processingOver()) {
            varArgChecker.scan(roundEnv.getRootElements(), null);}
        return false; /* Allow other processors to examine files too.*/ }

    @Override
    public void init(ProcessingEnvironment processingEnv) {
        super.init(processingEnv);
        this.messenger = processingEnv.getMessenger();
        varArgChecker = new VarArgCheckScanner();}

    private class VarArgCheckScanner extends ElementScanner6<Void, Void> {
        @Override
        public Void visitExecutable(ExecutableElement e, Void p) {
            List<? extends VariableElement> parameters = e.getParameters();
            if (!e.isVarArgs() && !parameters.isEmpty()) {
                VariableElement lastParam = parameters.get(parameters.size() -1);
                if (lastParam.asType().getKind() == ARRAY)
                    messenger.printMessage(NOTE, "Var-arg candidate", lastParam);
            }
            super.visitExecutable(e, p);
            return null;
        }
    }
}

```

Sample Output

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World.");
    }
    private void foo() {;}
    private void foo(int i, double d) {;}
}
```

```
> javac -processor VarArgProc -proc:only HelloWorld.java
HelloWorld.java:2: Note: Var-arg candidate
    public static void main(String[] args) {
                               ^
```

- See **javax.annotation.processing**
javax.lang.model.*

*When you come to a
fork in the road,
take it.
— Yogi Berra*

Of annotations and interfaces

- Problem:
Annotation processors need to provide various pieces of information to the processing framework, supported annotations, supported options, etc.
- Information is generally unchanging
- Provide information via annotations?
 - No, that is untyped
- Instead, impl. an interface with getter-methods
 - Standard way to provide functionality

Before — apt

```
public class ListClassApf implements AnnotationProcessorFactory {
    // Process any set of annotations
    private static final Collection<String> supportedAnnotations
        = unmodifiableCollection(Arrays.asList("*"));

    // No supported options
    private static final Collection<String> supportedOptions = emptySet();

    public Collection<String> supportedAnnotationTypes() {
        return supportedAnnotations;
    }

    public Collection<String> supportedOptions() {
        return supportedOptions;
    }

    public AnnotationProcessor getProcessorFor(
        Set<AnnotationTypeDeclaration> atds,
        AnnotationProcessorEnvironment env) {
        return new ListClassAp(env);
    }

    private static class ListClassAp implements AnnotationProcessor {
        private final AnnotationProcessorEnvironment env;
        ListClassAp(AnnotationProcessorEnvironment env) {
            this.env = env;
        }

        public void process() {
            System.out.println("Hello World.");
        }
    }
}
```

After further consideration...

- Chance to rethink **apt** design with JSR 269
- Boilerplate code is ugly
- Keep the interface with getter methods, *but*
 - Declare a helper class **AbstractProcessor**
 - Default implementation of **getFoo** reads the **@Foo** annotation on the subclass
- Type safety, but with convenience of annotations!
 - Subclasses can override default
 - Full flexibility of independent interface implementation
 - (At the cost of a few extra type declarations)

After — JSR 269

```
@SupportedAnnotationTypes({"*"}) // Process any set of annotations
public class ListClassProc extends AbstractProcessor {
    public boolean process(Set<? Extends Element> annotations,
                          RoundEnvironment roundEnv) {
        System.out.println("Hello World.");
        return false;
    }
}
```

Simplified Example

```
interface Processor {  
    String getFoo();  
}
```

```
@interface Foo() {  
    String value();  
}
```

```
public abstract class AbstractProcessor  
    implements Processor {  
    protected AbstractProcessor() {}  
    public String getFoo() {  
        String foo = this.getClass().getAnnotation(Foo.class);  
        return (foo == null ? "" : foo);  
    }  
}
```

```
@Foo("Bar")
```

```
public class MyProcessor extends AbstractProcessor{}
```

Compare

- **javax.annotation.processing
Processor
AbstractProcessor
Supported***
- **com.sun.mirror.apt
AnnotationProcessorFactory**

Of enums and interfaces

- Problem:
In JSR 199 (Java™ Compiler API) needed a notion of location, like the class path, to model where information was coming from or going to.
- An enum would be a natural way to capture this requirement *but*, the list of locations had to be extensible and enums are fixed.
- Could use a less structured, name based API

Do both!

- Declare an interface to capture the needed information (`JavaFileManager.Location`)
 - `getName`
 - `isOutputLocation`
- Declare an enum implementing the interface (`javax.tools.StandardLocation`)
 - Enums are classes too!
- API uses interface type for parameters, return
- Enum implementing companion interface also used forthcoming JSR 203 “Son of NIO” draft

Aside on equals and hashCode

- Care must be taken designing and implementing equals and hashCode with interfaces
 - Want working collections of the interface type
- Equals defined solely by contents of objects
 - All information for comparison must be provided by methods on the interface
 - hashCode must be defined over same information
- Equals *not* defined solely by contents of objects
 - Simplest: equality-is-identity (`.equals` is `==`), then no need to define hashCode :-)
 - Disjoint islands of implementations

Tasty filling and icing

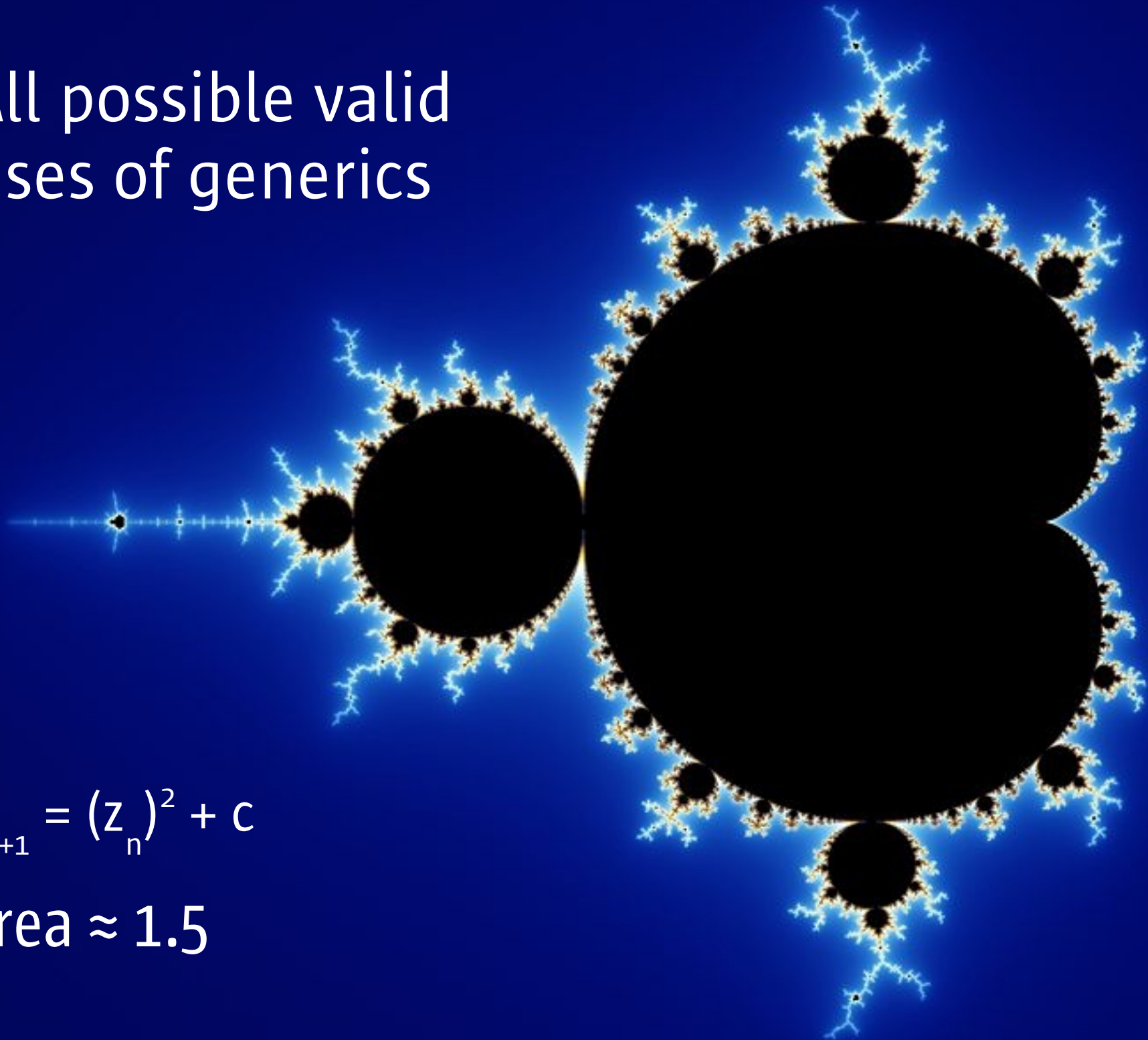
- Use interface to declare type
 - Capture functionality
 - Allow extensibility
- Use other language features to ease implementations

Generics

All possible valid uses of generics

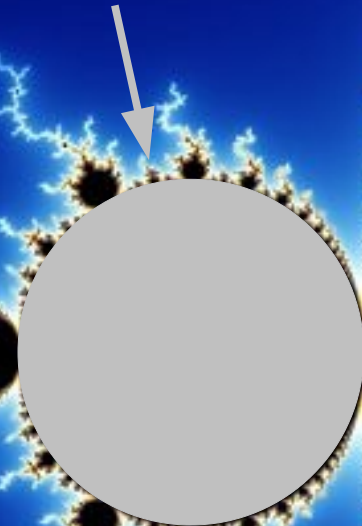
$$z_{n+1} = (z_n)^2 + c$$

Area ≈ 1.5

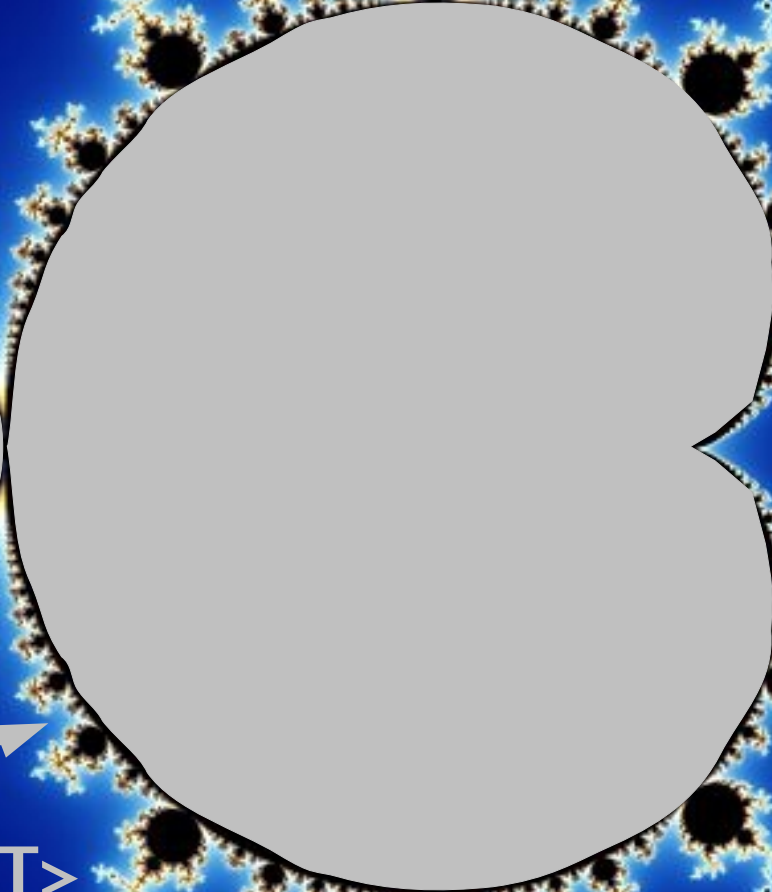


Simple uses of generics

Type tokens
`Class<T>`



Aggregates
`Set<T>`, `List<T>`



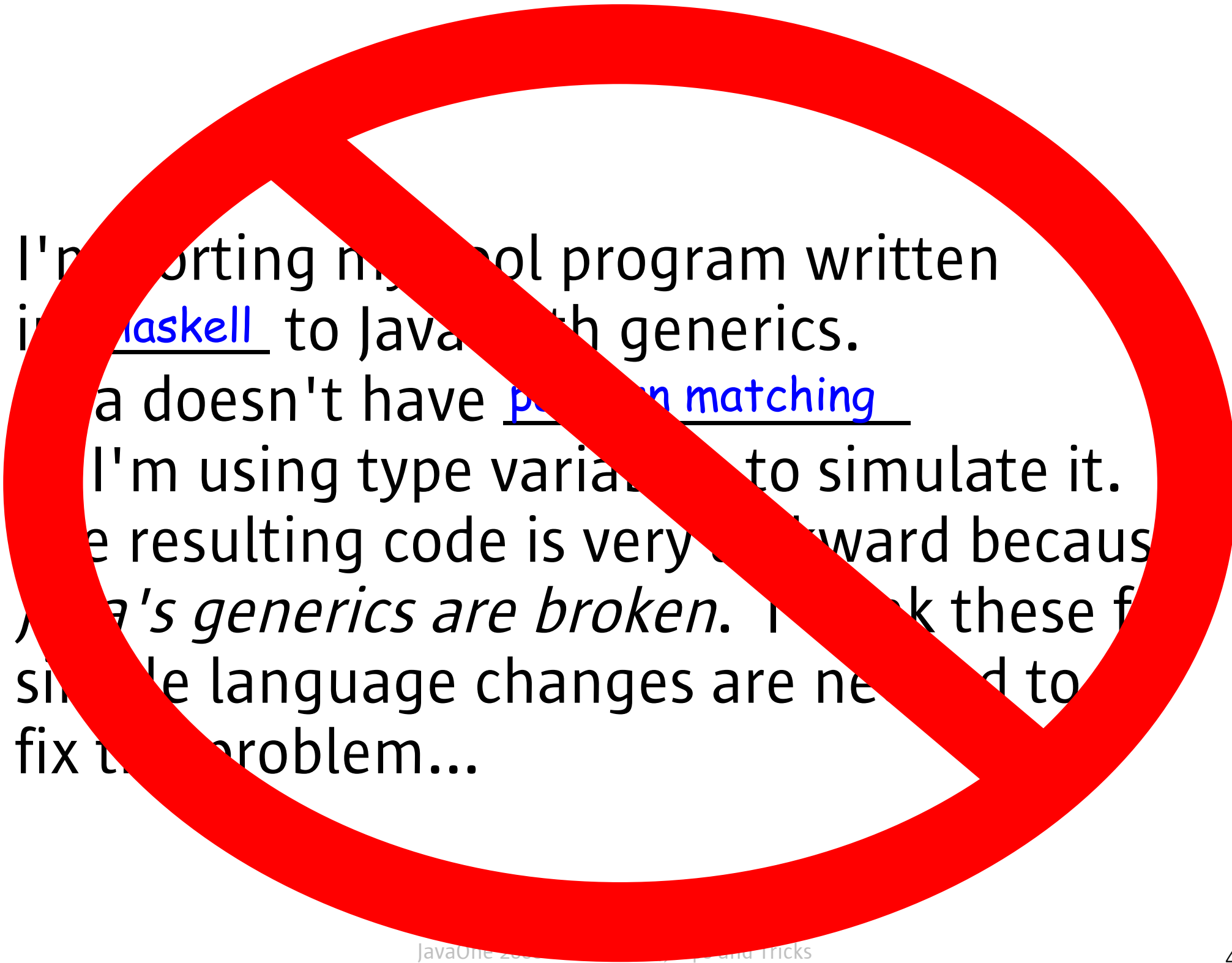
Area ≈ 1.37
91.4%

*Patient: Doctor, it hurts when I
move my arm like this.*

Doctor: Don't move your arm like that.

Anti-advice

I'm porting my cool program written in Haskell to Java with generics. Java doesn't have pattern matching so I'm using type variables to simulate it. The resulting code is very awkward because *Java's generics are broken*. I think these few simple language changes are needed to fix the problem...



I'm porting my tool program written
in Haskell to Java with generics.
Java doesn't have pattern matching
I'm using type variables to simulate it.
The resulting code is very awkward because
Java's generics are broken. I think these few
simple language changes are needed to
fix the problem...

Java Generics Aren't...

- ... C++ templates
 - No template meta-programming!
- ... necessarily a substitute for feature *X* in language *Y*
 - Idiomatic way to decompose a problem will vary by language and computational model
 - Cross-language syntax directed translations may suffer large impedance mismatches
- ... likely to be fun if your Java type is declared to have more than two type parameters

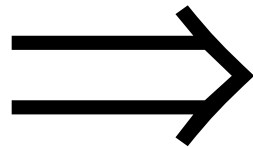
Generics: Some of the remaining 8.6%

Wildcards

Number



Integer



List<Number>

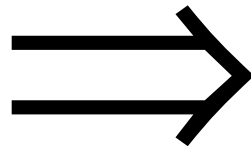


List<Integer>

Number



Integer



List<Number>



List<Integer>

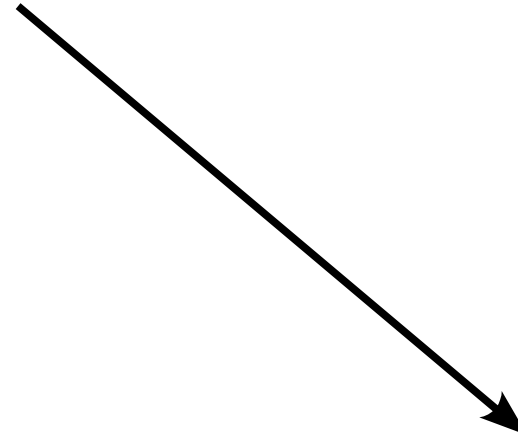
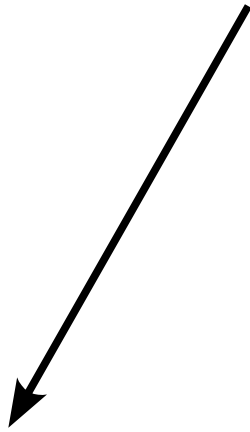
```
List<Number> listOfNumbers = listOfIntegers;  
listOfNumbers.add(BigDecimal.TEN);
```

Typing Relations

- What is relationship between **List<Number>** and **List<Integer>**
- Both are **java.lang.Objects** (lame!)
- Both are (raw) **java.util.Lists** (still lame!)
- Both are lists of *something*:
List<?>
- Both are lists of something that extends **Number**:
List<? extends Number>

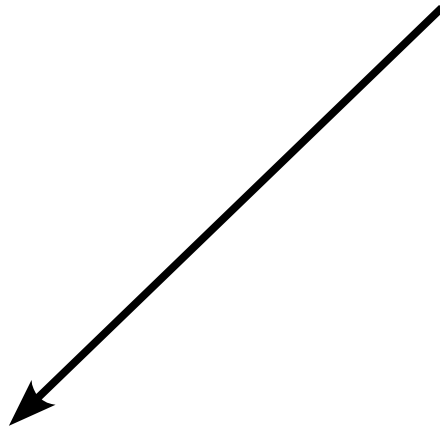
Subtyping with Wildcards

List<? extends Number>



List<? extends Integer>

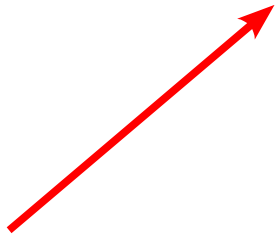
List<Number>



List<Integer>

Wait a minute...

```
List<? extends Number> listQENum = listQEIntegers;  
listOfNumbers.add(BigDecimal.TEN);
```



```
Test.java:6: cannot find symbol  
symbol   : method add(java.math.BigDecimal)  
location: interface java.util.List<capture#825  
                of ? extends java.lang.Number>  
listQEN.add(java.math.BigDecimal.TEN);
```

List<T> vs List<? extends T>

- Retrieved objects have the same bound:
`T element = listOfT.get(0);`
`T element = listOfQuestionExtendsT.get(0);`
- What can you do with the List itself:
`T element = listOfT.set(0, myT);`
`T element = listOfQuestionExtendsT.set(0, myT);`

... capture of #123 cannot be applied to ...



<T> void foo(S<T> s) vs void foo(S<?> s)

- In a generic method, if a type variable appears only once, it is usually clearer for the caller to deal with a wildcard
- If the type is needed, the wildcard can be *captured* using an internal private method with a type var.
- E.g. **Collections.rotate**

```
public void foo(S<?> s) {  
    return foo0(s);  
}
```

```
private <T> void foo0(S<T> s) {  
    ...  
}
```

Wildcards in JSR 269 return types

- Wildcards simpler for parameter types
- Less convenient as return type
 - Not so bad with for-each loop
- Specialized usage of wildcard return types in JSR 269
 - Not a mass-usage API!
 - Allows type safe return of internal collection implementations without copying
 - Allows specialized subinterfaces to be defined

Wildcards in JSR 269 return types, cont.

```
interface Type {  
    List<? extends Type> getParameterTypes()  
}  
  
class JavacType implements Type {  
    JavacList<JavacType> getParameterTypes() {  
        // No copy, no unsafe cast  
        return javacListOfJavacTypes;  
    }  
}
```

Wildcards in JSR 269 return types, cont.

```
interface Type {  
    List<? extends Type> getParameterTypes()  
}
```

```
interface TypeForMyLanguage extends Type {  
    List<TypeForMyLanguage> getParameterTypes();  
  
    int translucency();  
}
```

or

```
interface TypeForMyLanguage extends Type {  
    List<? extends TypeForMyLanguage> getParameterTypes();  
  
    int translucency();  
}
```

Miscellaneous

Be Wary of Interface Covariant Returns

- Return a more specific type in a subclass:
Relative: **Number foo()**;
Parent: **Integer foo()**;
Cousin: **Double foo()**;
- Very helpful in concrete implementation classes, e.g. `Foo clone()`;
- But if used in branching interface hierarchy, will prevent a single class from providing all methods in full tree/DAG of interfaces
 - May have unwelcome consequences, like wrapper objects

```
interface Relative {  
    Number foo();  
}
```

```
interface Parent  
    extends Relative {  
    Integer foo();  
}
```

```
class BachelorBob  
    implements Relative,  
             Cousin {  
    public Double foo() {  
        return 2.5;}  
}
```

```
interface Cousin  
    extends Relative {  
    Double foo();  
}
```

```
class UncleBuck  
    implements Cousin,  
             Parent {  
    public Double foo() {  
        return 1.0;}  
    public Integer foo() {  
        return 42;}  
}
```

UncleBuck.java:2: foo() in UncleBuck cannot implement foo() in Parent; attempting to use incompatible return type found : java.lang.Double required: java.lang.Integer
public Double foo() {

Tips on Exceptions

- If adding an Exception subclass, consider providing more information beyond a new name
 - What caused the exception?
 - How to recover?
- If adding multiple exceptions, consider having a common superclass
 - `javax.lang.model.element.UnknownFooException`
`javax.lang.model.element.UnknownBarException`
`javax.lang.model.type.UnknownBazException`
 - Should add superclass
`javax.lang.model.UnknownWhateverException`

Adding a parent

- All Exceptions are serializable so should have declared **serialVersionUID's**
 - (Use care defining getter methods)

//Before

```
UnknownFooException extends RuntimeException ...  
UnknownBarException extends RuntimeException ...  
UnknownBazException extends RuntimeException ...
```

// After

```
UnknownWhateverException extends RuntimeException ...  
UnknownFooException extends UnknownWhateverException ...  
UnknownBarException extends UnknownWhateverException ...  
UnknownBazException extends UnknownWhateverException ...
```

Summary

- Understand *precise* constraints for future changes
- Use interfaces along with other features to give flexibility for clients and implementors
- Favor simple generics
- Explore annotation processing for source analysis

Questions?

joe.darcy@sun.com

<http://blogs.sun.com/darcy>