

The slide features a decorative arrangement of seven circles. Three circles are filled with a light purple color, while four are hollow with a thin purple outline. They are arranged in two rows: three in the top row and four in the bottom row. The text is centered horizontally across the middle of the slide.

Meta Programming In Java

Mika Haapakorpi



Introduction

- **Meta-programming**

- The discipline of writing programs that represent and manipulate other programs or themselves [GP]

- **Java meta-programming mechanisms:**

- reflection

- generics

- metadata annotations

Reflection

Introduction

● Definition

- Ability to *introspect* metalevel information about the program structure itself at runtime.
 - Mechanisms to change the program interpretation or meaning at the runtime (*intercession*)
- Usually this metalevel information is modeled using the general abstraction mechanisms available in the language.

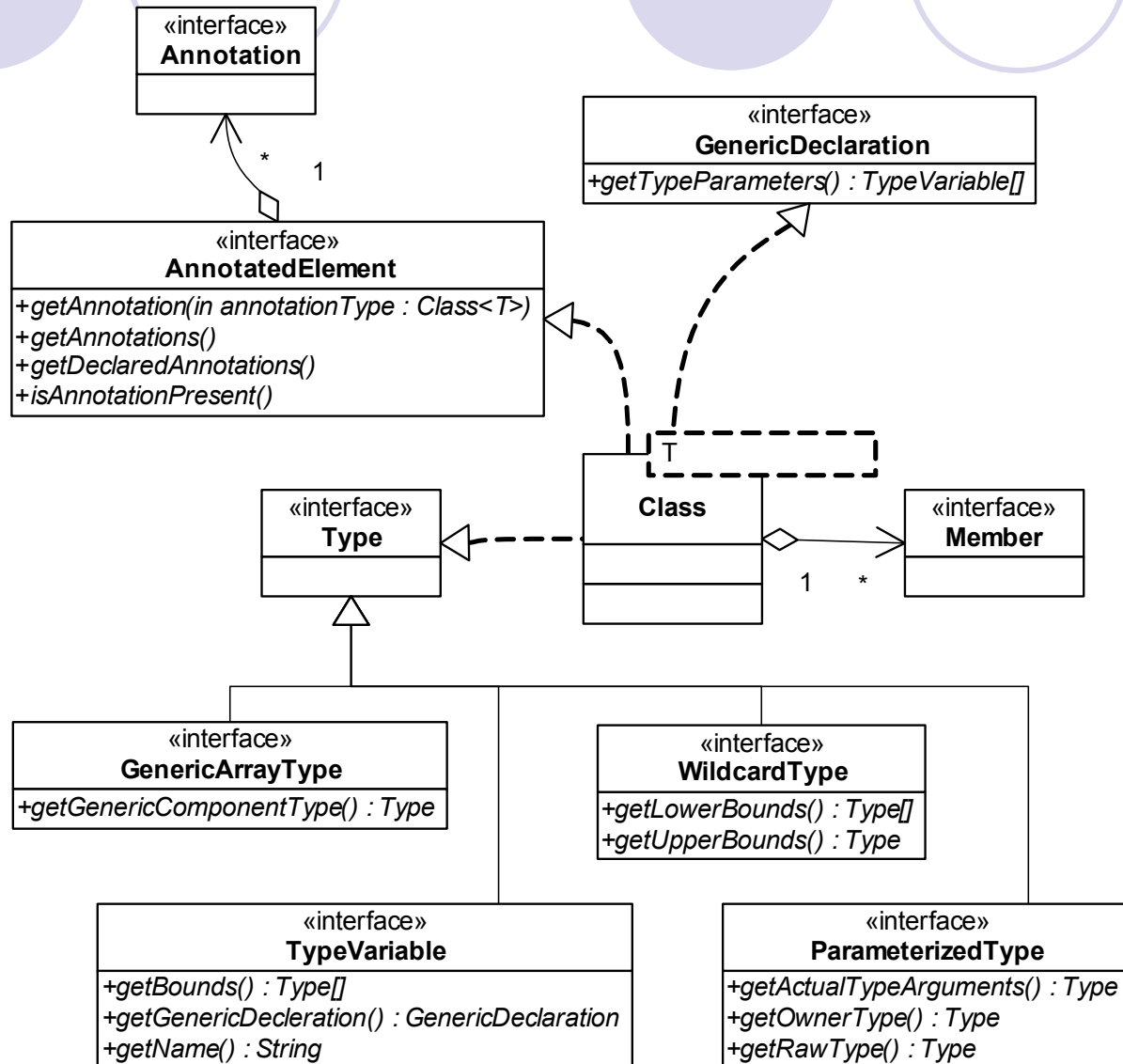
Reflection

Java

- In Java, reflection enables to discover information about the loaded classes:
 - Fields,
 - Methods, and
 - Constructors
 - Generics information
 - Metadata annotations
- It also enables to use these metaobjects to their instances in runtime environment.
 - E.g. `Method.invoke(Object o, Object... args)`

Reflection

Java metalevel architecture



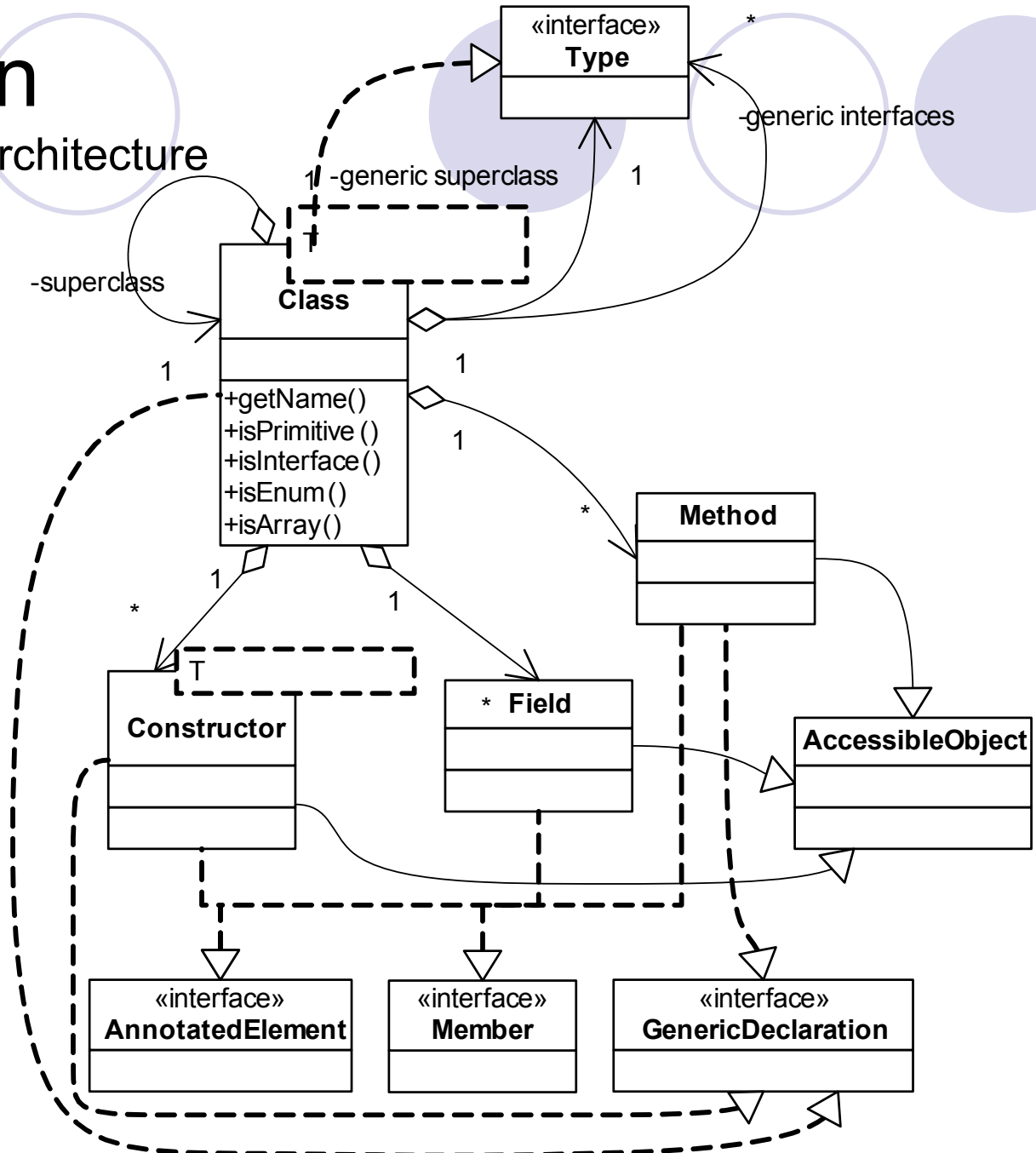
Reflection

Java metalevel architecture

- Type is the common superinterface for all types in the Java programming language.
 - Class/interface types (**Class**)
 - Array types (**Class**)
 - Enumeration types (**Class**)
 - Annotation types (**Class**)
 - Primitive types (**Class**)
 - Type variables (**TypeVariable**)
 - Parameterized types (**ParameterizedType**)
 - Generic array types (**GenericArrayType**)
 - Wildcard types (**WildcardType**)

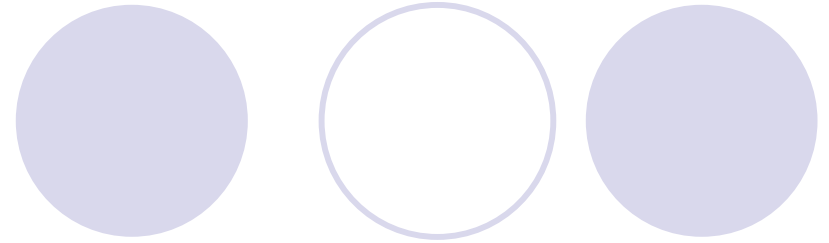
Reflection

Java metalevel architecture
Class



Reflection

Java metalevel architecture



- The `Class<T>` metaclass is parametrized over the generic `TypeVariable T`.
 - `T` represents any class or interface type i.e. the actual class, which is the instance of the metaclass `Class<T>`.
 - E.g. `T cast(Object o)`
- `Class<T>` itself is represented as a class and also has a corresponding metaclass `Class<Class>`

Example – The class of Class<T>

- Information about Class<T>
- ClassTest.java
 - Introspecting information about Class.class
 - Parametrized type (at compile time)
Class<Class>



Example –Class<ArrayList>

- Information about ArrayList class
- ListTest.java

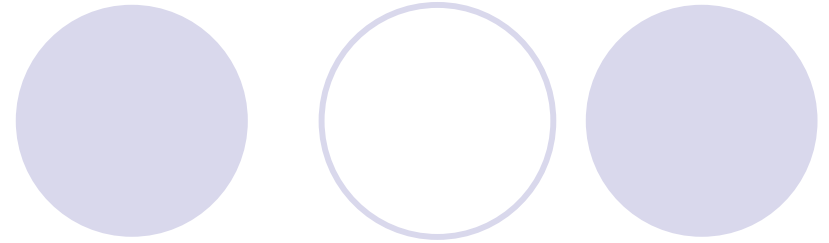
Reflection

Dynamic proxy classes

- Dynamic mechanism to create an implementation to some particular interface at runtime
- Basic Idea:
 - Associate an invocation handler to the set of interfaces
 - Every method call to the interface is dispatched to the invocation handler.

Reflection

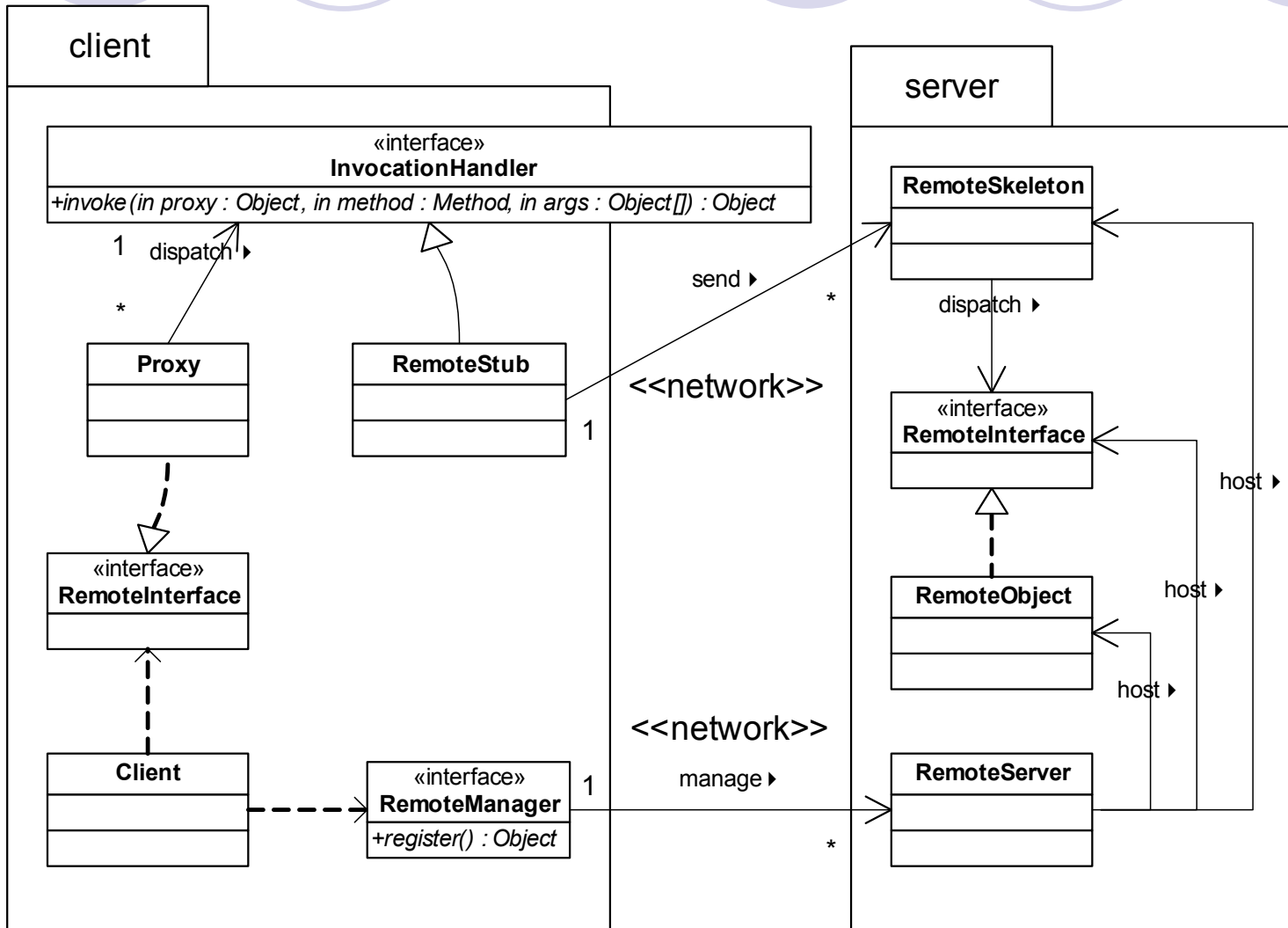
Dynamic proxy classes



- A *dynamic proxy class*
 - implements a list of interfaces specified at runtime when the class is created.
- A *proxy interface*
 - an interface that is implemented by a proxy class.
- A *proxy instance* is an instance of a *proxy class*.
 - has an associated *invocation handler* object.
- An *invocation handler*

Reflection

Dynamic proxy classes - example



Generics



Introduction

- Abstraction over types
- Generics are one of the new language features in J2SE 1.5. (Tiger) JSR-176
- The generics are specified in the JSR-014 [Gen].
 - i.e. the generics are developed under JCP
 - www.jcp.org
- Resembles the template mechanism in C++
 - But these are NOT templates

Generics



- The generics can be used in classes, interfaces, methods and constructors.
- Two new types:
 - Parametrized types
 - Type variables
- A type variable is an unqualified identifier.
- Class and interface declarations can have type arguments (type variables)
 - Metalevel: Class implements GenericsDeclaration
- Method and constructors definitions can have type arguments (type variables)
 - Metalevel: Method, Constructor implements GenericsDeclaration

Generics

Type syntax

```
ReferenceType ::= ClassOrInterfaceType  
    | ArrayType  
    | TypeVariable
```

```
TypeVariable ::= Identifier
```

```
ClassOrInterfaceType ::= ClassOrInterface  
    TypeArgumentsOpt
```

```
/* includes parametrized version */
```

```
ClassOrInterface ::= Identifier
```

```
    | ClassOrInterfaceType . Identifier
```

```
TypeArguments ::= < ReferenceTypeList >
```

```
ReferenceTypeList ::= ReferenceType
```

```
    | ReferenceTypeList , ReferenceType
```

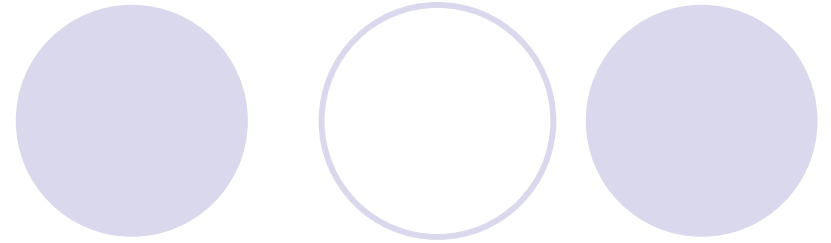
Generics

Types - example

- `List<String> anExample = new ArrayList<String>()`
 - `List` = interface
 - `ArrayList` = class
 - `String` = class (the actual type argument)
 - `List<String>` and `ArrayList<String>` = parametrized type
 - No runtime support
 - `anExample.getClass() == ArrayList.getClass()`

Generics

Types - examples



- GenericsTest.java

Generics

Class/Interface Declarations

ClassDeclaration ::= ClassModifiersOpt class Identifier
TypeParametersOpt

SuperOpt InterfacesOpt ClassBody

InterfaceDeclaration ::= InterfaceModifiersOpt interface
Identifier TypeParametersOpt

ExtendsInterfacesOpt InterfaceBody

TypeParameters ::= < TypeParameterList >

TypeParameterList ::= TypeParameterList , TypeParameter
| TypeParameter

TypeParameter ::= TypeVariable TypeBoundOpt

TypeBound ::= extends ClassOrInterfaceType
AdditionalBoundListopt

AdditionalBoundList ::= AdditionalBound
AdditionalBoundList

| AdditionalBound

AdditionalBound ::= & InterfaceType

Generics

Examples

```
public interface List<E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

```
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

```
List<String> anExample;
```

```
anExample.add("sdfdfss");
```

```
anExample.add(new Object()); // compile time error
```

```
String aTest = anExample.iterator().next();
```

Generics

Erasure

- Erasure erases all generics type argument information at compilation phase.
 - E.g. `List<String>` is converted to `List`
 - E.g. `String t = stringlist.iterator().next()` is converted to `String t = (String) stringlist.iterator().next()`
- As part of its translation, a compiler will map every parameterized type to its type erasure.

Generics

Erasure

```
List <String> l1 = new  
    ArrayList<String>();
```

```
List<Integer> l2 = new  
    ArrayList<Integer>();
```

```
System.out.println(l1.getClass()  
    == l2.getClass());
```

Generics



Type safety

- Type safety: even when using generic classes like `ArrayList<T>`
 - It is possible to create bytecode that throws `ClassCastException`
 - Create source code that throws `ClassCastExceptions` (raw types)
 - Create source code using reflection that throws `ClassCastException` (metaclasses does not contain the type arguments)
- However
 - If your entire application has been compiled without unchecked warnings using `javac -source 1.5`, it is type safe.
 - Provided that you do not have manual casts and do not use reflection.
 - The type safety and integrity of the Java virtual machine are never at risk, even in the presence of unchecked warnings.

Generics

Loop hole – raw types

- Source code:

```
public String loophole(Integer x) {  
    List<String> ys = new LinkedList<String>();  
    List xs = ys; // raw type  
    xs.add(x); // compile-time unchecked warning  
    return ys.iterator().next();  
}
```

- Runtime behaviour:

```
public String loophole(Integer x) {  
    List ys = new LinkedList;  
    List xs = ys;  
    xs.add(x);  
    return (String) ys.iterator().next(); // run time error  
}
```

Generics



Methods

- Method/Constructor declarations can be generic
- Type arguments are not passed to generic methods when they are used

Generics

Methods - Examples

```
static <T> void fromArrayToCollection(  
    T[] a, Collection<T> c)  
{  
    for (T o : a) {  
        c.add(o); // correct  
    }  
}  
Object[] oa = new Object[100];  
Collection<Object> co = new ArrayList<Object>();  
  
Number[] na = new Number[100];  
Collection<Number> cn = new ArrayList<Number>();  
  
String[] sa = new String[100];  
Collection<String> cs = new ArrayList<String>();  
  
fromArrayToCollection(oa, co); // T inferred to be Object  
fromArrayToCollection(na, co); // T inferred to be Object  
fromArrayToCollection(na, cs); // compile-time error
```

Annotations



- Developers can define custom *annotation types*
- Using these types developers can annotate
 - fields,
 - methods,
 - classes,
 - and other program elements.

Annotations



- Annotation does not effect the program semantics
- Motivation: Development tools can read these annotations and generate new artifacts accordingly
 - Source files
 - Configuration files
 - XML documents

Annotations



- Tools can read the annotations from source files or from the class files
- For example, a tool called a *stub generator* could generate remote procedure call stubs as directed by annotations indicating which methods were designed for remote use.
- Previous annotation like mechanisms (JavaDoc)
 - @deprecated
 - @author
 - @version

Annotations

Declarations

AnnotationTypeDeclaration:

```
InterfaceModifiersopt @ interface Identifier AnnotationTypeBody
```

AnnotationTypeBody:

```
{ AnnotationTypeMemberDeclarationsopt }
```

AnnotationTypeMemberDeclarations:

```
AnnotationTypeMemberDeclaration
```

```
AnnotationTypeMemberDeclarations AnnotationTypeMemberDeclaration
```

AnnotationTypeMemberDeclaration:

```
AbstractMethodModifiersopt Type Identifier ( ) DefaultValueopt ;
```

```
ConstantDeclaration
```

```
ClassDeclaration
```

```
InterfaceDeclaration
```

```
EnumDeclaration
```

```
AnnotationTypeDeclaration
```

```
;
```

DefaultValue:

```
default MemberValue
```

Annotations

Declarations

- Form of a highly restricted interface declaration:
 - No extends clause is permitted.
 - Annotation types automatically extend a marker interface, `java.lang.annotation.Annotation`.
 - Methods must not have any parameters.
 - Methods must not have any type parameters
 - Generic methods are prohibited.
 - Method return types are restricted to
 - primitive types, `String`, `Class`, enum types, annotation types, and arrays of the preceding types.
 - No throws clause is permitted.
 - Annotation types must not be parameterized
- Legal wherever interface declarations are legal, and have the same scope and accessibility.

Annotations

Declaration - Examples

```
/**
 * Annotation with this type indicates that
 * the specification of the annotated API element
 * is preliminary and subject to change. (Marker annotation)
 */
public @interface Preliminary { }

/**
 * Associates a copyright notice with the
 * annotated API element. (Single member annotation)
 */
public @interface Copyright { String value(); }

/**
 * A person's name. This annotation type is not designed to be used
 * directly to annotate program elements, but to define members
 * of other annotation types. (Complex annotation)
 */
public @interface Name {
    String first();
    String last();
}
```

Annotations



- Annotations may be used as modifiers in any declaration
 - class, interface, field, method, parameter, constructor, enum, or local variable.
 - package declarations
- Annotation types are imported in the same fashion as classes and interfaces.

Annotations



Examples

```
// Marker annotation
```

```
@Preliminary public class TimeTravel { ... }
```

```
// Single-member annotation
```

```
@Copyright("2004 Solita, Inc., All rights reserved.")  
public class Organization { ... }
```

```
// Single-member complex annotation
```

```
@Author(@Name(first = "Joe", last = "Hacker"))  
public class BitTwiddle { ... }
```

Annotations

Examples

```
/**
 * Designates a formatter to pretty-print the annotated
 * class.
 */
public @interface PrettyPrinter {
    Class<? extends Formatter> value();
}

// Single-member annotation with Class
// member restricted by bounded wildcard
// The annotation presumes the existence of this class.
class GorgeousFormatter implements Formatter { ... }

@PrettyPrinter(GorgeousFormatter.class)
public class Petunia { ... }
```