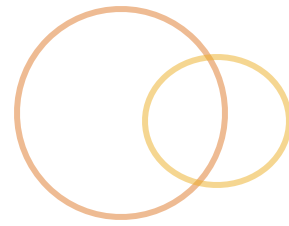


Advanced Language Features



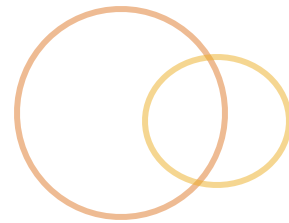
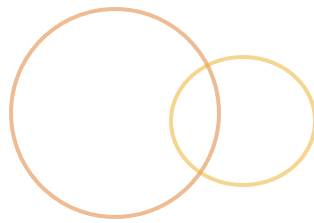
Presentation Topics



In this section, we will cover:

- 🕒 Type-safe Enumerations
- 🕒 Generics
- 🕒 Metadata
- 🕒 Covariant Returns

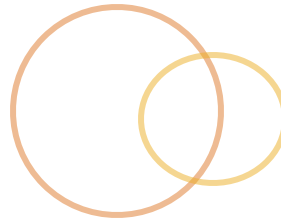
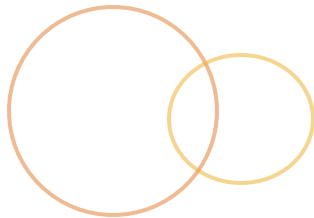
Objectives



When we are done, you should be able to:

- 🕒 Create a simple enumeration
- 🕒 Incorporate generics into “legacy” code
- 🕒 List 2 annotations

Type-Safe Enumerations



Type-Safe Enumerations

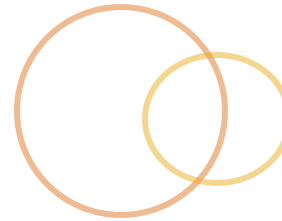
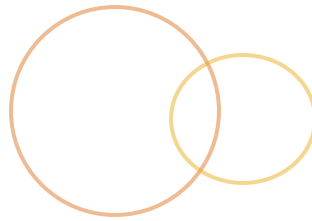
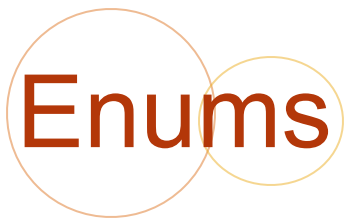


What is an Enumeration?

- Comes from mathematical world
- Represents finite listing of values

What is a type-safe enumeration?

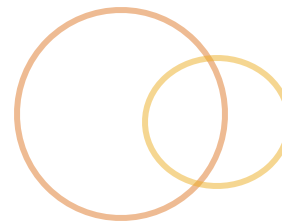
- Language-based mechanism to represent finite listing
- Represents a collection of typed-values
- Immutable



Why do they exist?

- Historically implemented using an enum pattern
- Common problems with enum pattern:
 - Not type-safe
 - No separate namespace (values typically defined as fields)
 - Based on primitive values that may change
- Laborious to develop using enum pattern
 - Creates code level dependencies
 - Tons of boiler-plate code

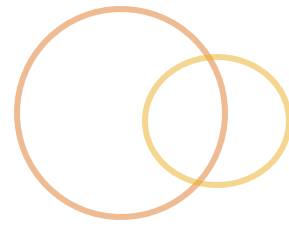
Enums [cont.]



⦿ How do they work?

- ⦿ Look similar to enumeration support in other languages
- ⦿ Considered new type, enum type
- ⦿ Full-fledged type support:
 - ⦿ Fields
 - ⦿ Methods
 - ⦿ Constructors
- ⦿ Support Object level functionality like:
 - ⦿ Comparison
 - ⦿ Serialization
 - ⦿ `toString`, `equals`, etc.

Creating an Enum



- ◎ Two ways to create an enum
 - ◎ Top-level type declaration
 - ◎ Inner-class type declaration
- ◎ In both cases:
 - ◎ Declare enum type
 - ◎ Define with “values”

Top-level Enum Example



```
1  package examples.enums;
2
3  /**
4   * Days is a basic illustration of an
5   * enumerated type within the Java language.
6   */
7  public enum Days {
8      SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
9      THURSDAY, FRIDAY, SATURDAY;
10 }
11
```

Inner-class Enum Example



```
1  package examples.enums;
2
3  /** ... */
7  public class Calendar {
8      public enum Days { SUNDAY, MONDAY, TUESDAY,
9                          WEDNESDAY, THURSDAY, FRIDAY,
10                         SATURDAY };
11  }
12
```

Working with an Enum



- ◎ Enums are types
- ◎ Values are instances of an enum type
 - ◎ Stored as `static final` fields in type
 - ◎ Defined in terms of
 - ◎ `name` - stringified representation of field name
 - ◎ `ordinal` - position in set
 - ◎ Referencable through dot-notation
 - ◎ Are switchable

Accessing an Enum Value Example



```
1 package examples.enums;
2
3 /** ... */
7 public class DaysExample {
8
9     public static void main(String[] args) {
10         Days today = Days.SUNDAY;
11         System.out.println("Today is: " + today);
12     }
13 }
14
```

Enum Switch Example



```
1 package examples.enums;
2 /** ... */
7 public class DaysSwitchExample {
8
9     public static void main(String[] args) {
10         Days today = Days.SUNDAY;
11         String message = getMessage(today);
12         System.out.print("Today is " + today);
13         System.out.println(", I should go " + message);
14     }
15
16     private static String getMessage(Days today) {
17         String message;
18         switch(today) {
19             case SATURDAY:
20                 message = "play";
21                 break;
22             case SUNDAY:
23                 message = "to church";
24                 break;
25             default:
26                 message = "work";
27                 break;
28         }
29         return message;
30     }
31 }
32
```

Working with an Enum [cont.]



- ⦿ Enums have some predefined *static* methods
 - ⦿ `values` — retrieves all enum instances
 - ⦿ `valueOf` - transforms `String` value into enum instance
- ⦿ Have some predefined *instance* methods
 - ⦿ `name` — upper-case name of enum instance
 - ⦿ `toString`
 - ⦿ `equals`
 - ⦿ `hashCode`

Enum Method Example



```
1 package examples.enums;
2
3 /**...*/
9 public class DaysValuesExample {
10
11     public static void main(String[] args) {
12         for(Days d : Days.values())
13             System.out.println(d.name());
14     }
15
16 }
17
```

Prints:

SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY

Working with an Enum [cont.]



- ⦿ Enums can have methods
- ⦿ Accessed using dot-notation
- ⦿ Can have static methods associated with enum
- ⦿ Can have instance methods associated with enum values

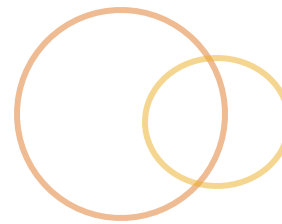
Enum Method Example



```
1 package examples.enums;
2
3 /** ... */
4
5 enum Days {
6     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
7     THURSDAY, FRIDAY, SATURDAY;
8
9     public String getReadableName() {
10         //get name as String
11         String nameValue = name();
12
13         //convert it to correct capitalization
14         return nameValue.substring(0, 1) +
15             nameValue.substring(1).toLowerCase();
16     }
17 }
18
19 }
```

```
1 package examples.enums;
2
3 /** ... */
4
5 public class DaysMethodExample {
6
7     public static void main(String[] args) {
8         Days today = Days.SUNDAY;
9         System.out.println("Today is: " + today.getReadableName());
10     }
11 }
12
13 }
```

Enum Lab 1



🕒 Description:

Create an enumeration called `Month` to represent the months of the year. Allow the user to specify their favorite month from the command line. Convert the `String` value for the month to the appropriate `Month` enum value. If an invalid month is specified, notify the user and print off all valid values for `Month`. Once a `Month` is selected, print off a message describing what season the month belongs to.

🕒 Duration: 20 minutes

Working With an Enum [cont.]



- ⦿ Enums support method overriding
 - ⦿ Enum-defined methods
 - ⦿ Object methods
- ⦿ Method over-riding supported:
 - ⦿ Across all enum instances
 - ⦿ Specific instance

Instance Method Overriding Example



```
1 package examples.enums;
2
3 /** ... */
9 public class DaysValuesExample2 {
10
11     public static void main(String[] args) {
12         for(Days d : Days.values())
13             System.out.println(d.getReadableName());
14     }
15
16 }
17
```

```
1 package examples.enums;
2
3 /** ... */
7 enum Days {
8     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
9     THURSDAY, FRIDAY,
10    SATURDAY { //treat saturday different
11        public String getReadableName() {
12            return name();
13        }
14    };
15
16    public String getReadableName() {
17        //get name as String
18        String nameValue = name();
19
20        //convert it to correct capitalization
21        return nameValue.substring(0, 1) +
22            nameValue.substring(1).toLowerCase();
23    }
24 }
25
```

Enum Method Overriding Example



```
1 package examples.enums;
2
3 /**...*/
7 enum Days {
8     SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
9     THURSDAY, FRIDAY,
10    SATURDAY { //treat saturday different
11        public String getReadableName() {...}
14    };
15
16    public String getReadableName() {...}
24
25    public String toString() {
26        return getReadableName();
27    }
28 }
29
```

```
1 package examples.enums;
2
3 /**...*/
9 public class DaysValuesExample3 {
10
11    public static void main(String[] args) {
12        for(Days d : Days.values())
13            System.out.println(d);
14    }
15
16 }
17
```

Working with an Enum [cont.]



- ⦿ Enums support constructors
- ⦿ Constructors are private
- ⦿ Used to initialize instance variables
- ⦿ Provide type-safe instance creation

Enum Constructor Example



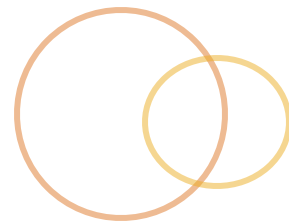
```
1 package examples.enums;
2
3 /** ... */
4
5 enum DaysToo {
6     SUNDAY ("Sunday"), MONDAY ("Monday"),
7     TUESDAY ("Tuesday"), WEDNESDAY ("Wednesday"),
8     THURSDAY ("Thursday"), FRIDAY ("Friday"),
9     SATURDAY;
10
11     private String readableName;
12
13     DaysToo() {
14         readableName = name();
15     }
16
17     DaysToo(String s) {
18         readableName = s;
19     }
20
21     public String getReadableName() {
22         return readableName;
23     }
24
25     public String toString() {
26         return getReadableName();
27     }
28 }
29
30
```

Advanced Enum Features



- ◎ Enums are types
 - ◎ No enum - enum inheritance chains
 - ◎ No class - enum inheritance chains
 - ◎ Can implement interfaces
- ◎ Two new enumeration oriented collections
 - ◎ EnumMap— converts enum fields into map keys
 - ◎ EnumSet— converts enum fields into a set

Enum Lab 2

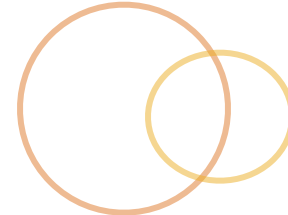
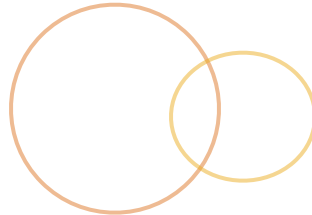


🕒 Description:

Modify Enum Lab 1. Refactor the enumeration implementation, adding a more robust static factory method. Regardless of the case of the `String` passed in, if the `String` matches a `Month`'s name, the `Month` should be returned. Additionally, associate a `Season` with each enum value and make it accessible through a `getSeason` method call. Modify your application to use the enum modifications.

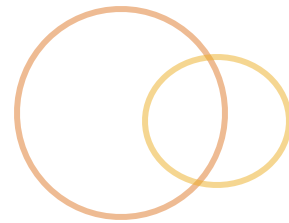
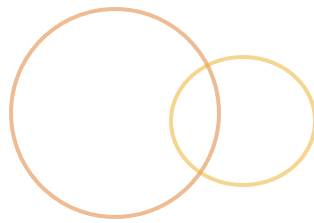
🕒 Duration: 20 minutes

Generics



Black and white never tasted so good

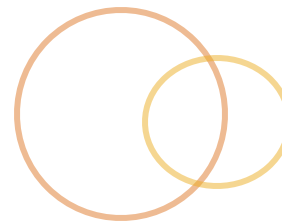




What are generics?

- Stands for generic types and generic methods
- Represent design pattern known as *parameterized types* and methods
- Allows a type to be defined without specifying all of the other types it uses
- Were one of most requested features of language

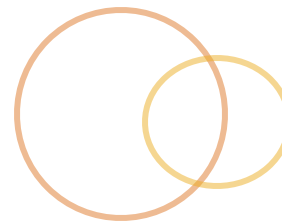
Generics [cont.]



Why do they exist?

- Add type-awareness to collections, without breaking flexibility
- Add type-awareness to other container-like classes, without breaking flexibility
- Add type awareness to methods
- Provide compile-time type-safety
 - Remove development-time casting procedures
 - Remove run-time type incompatibilities
 - Remove run-time `ClassCastException`s

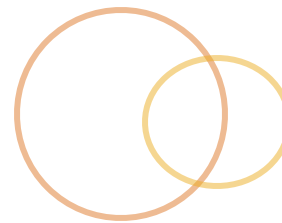
Using Generics



⦿ How do they work?

- ⦿ Supports both definition and application
 - ⦿ Most straightforward is application
 - ⦿ But application requires understanding definition
- ⦿ Use “placeholder” to represent generic type as part of type or method definition
 - ⦿ Placeholder value is replaced with type in source
 - ⦿ Placeholder is removed during compilation, replaced with traditional casting (known as type erasure)

Generic Placeholders



Generic type placeholders

- Used when defining a parameterized type

- `<E>` - stands for element; represents element type held within container

- `<T>` - stands for type

- `<V>` - stand for value

Generic method placeholders

- Used when defining a parameterized method

- `<E>` - parameterized type

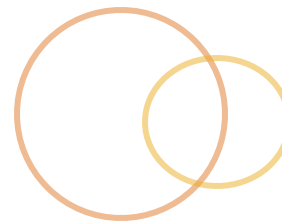
- `<?>` - wildcard placeholder

- `<? extends E>` - bounded wildcard placeholder

- `<? super E>` - bounded wildcard placeholder

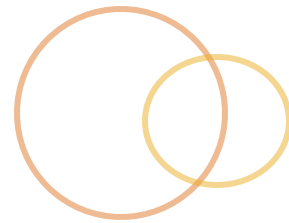
- `<E>`, `<T>`, `<V>`, etc. naming convention only

Generic Collections



- Collections API has been rewritten to support Generics
 - Provides type safety to collections
 - Applies to *all* classes within Collection API
- Specify the type the collection will hold
 - Inserting type mismatch generates compile-time error
 - Getting / removing element no longer requires cast
- Backwards compatible in *raw type* format
 - May generate compile-time warning
 - Can widen typed collection into raw-type

Generic List

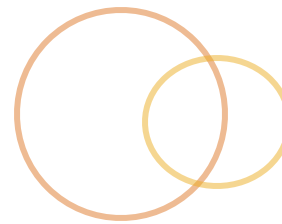


- ◎ List represents an ordered collection
- ◎ List interface now represents generic type

```
public interface List<E> extends Collection<E> {..}
```

- ◎ Read as *List of <type E> elements*
- ◎ Certain List methods now generic
 - ◎ `Iterator<E> iterator();`
 - ◎ `boolean containsAll(Collection<?> c);`
 - ◎ `boolean addAll(Collection<? extends E> c);`

Generic ArrayList



- Provides type-safe representation of an array-backed list
 - Implementation of `List` interface
 - Subclass of `AbstractList`
 - Common replacement for `Vector`
- Create `ArrayList` using parameterized syntax:
 - `List<String> myList = new ArrayList<String>();`
 - `<String>` replaces placeholder `<E>`
 - Read as *List of String elements*
 - `myList` can only hold `String` elements

Simple List Example [Old way]



```
1  package examples.generics.simple;
2  +import ...
6  +/**...*/
11 public class OldWayExample {
12
13     public static void main(String[] args) {
14         List myList = new ArrayList();
15         //convert args into a List
16         List argList = Arrays.asList(args);
17         //add Strings to list
18         myList.addAll(argList);
19         //list is not typesafe, can add any object
20         myList.add(new Integer(0));
21
22         Iterator theArgs = myList.iterator();
23
24         //step through list elements
25         while (theArgs.hasNext()) {
26             //will cause class cast
27             // exception with Integer element
28             String nextArg = (String) theArgs.next();
29         }
30     }
31 }
32
```

Simple List Example [New way]



```
1  package examples.generics.simple;
2  +import ...
6  +/**...*/
10 public class TestExample {
11
12     - public static void main(String[] args) {
13         //typesafe List of String elements
14         List<String> myList = new ArrayList<String>();
15
16         //convert args into a List<String>
17         List<String> argList = Arrays.asList(args);
18         myList.addAll(argList);
19
20         //would cause compile-time error
21         //myList.add(new Integer(0));
22
23         //Iterator is now also typesafe
24         Iterator<String> theArgs = myList.iterator();
25         while(theArgs.hasNext()) {
26             String nextArg = theArgs.next();
27         }
28     }
29 }
30
```

Typesafe Collection Advantages



- ⦿ Adds compile time type safety
 - ⦿ `OldWayExample` allowed `Integer` to be inserted into collection; discovered problem at run-time
 - ⦿ `TestExample` prevented `Integer` to be inserted into collection; discovered at compile-time
- ⦿ Simplified interactions
 - ⦿ `OldWayExample` required casting when working with collection elements
 - ⦿ `TestExample` contained specific type; so no casting needed
- ⦿ No advantages in speed or performance

How Do They Work? [revised]



- ⦿ Implemented different than other languages
 - ⦿ Adopt ***type erasure*** mechanism
 - ⦿ Parameterized placeholder replaced at compile time
 - ⦿ Code converted from parameterized to generic
 - ⦿ Compiler “inserts” cast similar to `OldWayExample`
 - ⦿ Compiler ensures type-safety
 - ⦿ Only at compile time
 - ⦿ Run-time relies on traditional mechanism
 - ⦿ As a result, can still encounter run-time exceptions

Simple List Example [corrupted]



```
1  package examples.generics.simple;
2  +import ...
6  +/**...*/
10 public class CorruptTestExample {
11
12     public static void main(String[] args) {
13         //typesafe List of String elements
14         List<String> myList = new ArrayList<String>();
15
16         //convert args into a List<String>
17         List<String> argList = Arrays.asList(args);
18         myList.addAll(argList);
19
20         //call third-party api which uses raw types
21         ThirdPartyAPI.addElement(myList);
22
23         //Iterator is now also typesafe
24         Iterator<String> theArgs = myList.iterator();
25         while(theArgs.hasNext()) {
26             String nextArg = theArgs.next();
27         }
28     }
29 }
30
```

Simple List Example [corrupted]



- Third-party API does not utilize type-safe collections
- Causes issues at run-time (adds an Integer)

```
1  package examples.generics.simple;
2
3  import java.util.List;
4
5  /**...*/
11 public class ThirdPartyAPI {
12
13     public static void addElement(List list) {
14         list.add(new Integer(32));
15     }
16 }
17
```

Solidifying Type-safety



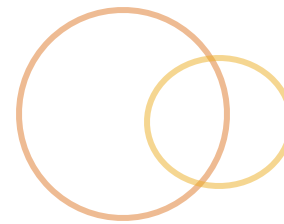
- ◎ Type erasure can be “stepped” around
- ◎ Should be a facility to guarantee type-safety, even with older / third-party APIs
- ◎ Collection facility adds wrappers to increase safety
 - ◎ Does not guarantee type-safety
 - ◎ But prevents insertion of type mismatched objects
 - ◎ Encounter `ClassCastException` on insertion instead
 - ◎ Theoretically easier to debug

Solidifying Type-safety [cont.]



- Dynamic type-safety support provided by collections class
 - Collections class rewritten to support generics
 - New static methods used to create a “checked” collection
 - ```
public static <E> List<E> checkedList(
 List<E> list, Class<E> type);
```
    - ```
public static <K, V> Map<K, V> checkedMap(  
    Map<K, V> m, Class<K> keyType,  
    Class<V> valueType);
```
 - Similar to other static methods used to create things like synchronized collections

Generics Lab 1



- 🕒 Description: Use the `Mixer` as your starting point. Refactor the `Mixer` so that the frequency map becomes type safe. The frequency map should contain `<String, Integer>` as its map structure. The `List` should contain `<String>` as its element types. Validate that you have written a type-safe `Mixer` using the `-Xlint` option with the compiler.
- 🕒 Duration: 15 minutes

Working with Generics



- ⦿ Generics go beyond type-safe collections
- ⦿ Can create generic methods
- ⦿ Can create own generic types

Creating Generic Methods



- ◎ Relatively straightforward process
 - ◎ Can add generic method support to any class
 - ◎ Use when you want to place type constraints on method
 - ◎ Simply add generic method nomenclature to method signature
 - ◎ Adjust method parameter list
 - ◎ Adjust method return signature

Revised Third-party Example



- Can modify method signature to ensure compile-time type-safety
- Only applicable if have “third-party” code

```
1 package examples.generics.simple;
2
3 import java.util.List;
4
5 /** ... */
10 public class TypeSafeThirdPartyAPI {
11
12     public static void addElement(List<String> list) {
13         list.add("Hello Typesafety");
14     }
15 }
16
```

Creating Generic Methods [cont.]



Can get fancy with wildcarding

- Adds flexibility to method signature

- Can be confusing

- `<?>` - unknown type

- unbounded wildcard

- use when you don't know or care about the value's type; like raw types

- `<? extends Number>`

- Upper-bounded wildcard

- Specified type should be `Number` or any subclass of `Number`

- `<? super Number>`

- Lower-bounded wildcard

- Specified type should be a direct subclass of `Number`

Bounded Third-party Example



```
1 package examples.generics.advanced;
2
3 import java.util.List;
4
5 /**
6  * The following class represents
7  * a Third-party API that can
8  * maintains a bounded type-safe collection
9  */
10 public class BoundedTypeSafeThirdPartyAPI {
11
12     public static void addElement(List<? super Number > list) {
13         list.add(729);
14         list.add(Math.PI);
15     }
16 }
17
```

Creating Generic Type



- Relatively straight-forward process
 - Create generic type like any type
 - Include generic type nomenclature
 - Reference placeholder within code
 - Have methods support generic type
- Could be used for things like:
 - Custom data structure
 - Generic value object

GenericVO Example



```
1 package examples.generics.advanced;
2
3 public class GenericVO<A,B> {
4
5     A fieldA;
6     B fieldB;
7
8     GenericVO(A a, B b) {
9         fieldA = a;
10        fieldB = b;
11    }
12
13    public void setFieldA(A a) {
14        this.fieldA = a;
15    }
16
17    public A getFieldA() {
18        return fieldA;
19    }
20
21    public void setFieldB(B b) {...}
24    public B getFieldB() {...}
27 }
28
```

GenericVO Example [cont.]



```
1 package examples.generics.advanced;
2
3 public class GenericVOExample {
4
5     public static void main(String[] args) {
6         //create instances of the GenericVO
7         GenericVO<String, String> name =
8             new GenericVO<String,String>("John", "Doe");
9
10        GenericVO<String, Integer> user =
11            new GenericVO<String,Integer>("john_doe123", 123457);
12
13        //get name field <B>
14        String lastName = name.getFieldB();
15        System.out.println("name's field <B> is: " + lastName);
16
17        //get user field <B>
18        Integer userId = user.getFieldB();
19        System.out.println("users's field <B> is: " + userId);
20    }
21
22 }
23
```

APIs Effected by Generics



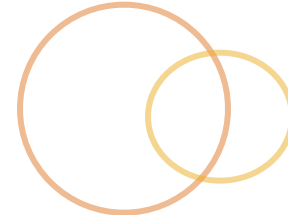
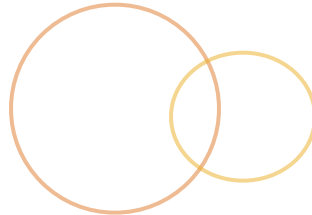
- 🕒 Collections API
- 🕒 Reflection API
- 🕒 Concurrency API
- 🕒 `java.lang` classes - like `Comparable`

Constructing Generics Lab



- 🕒 **Description:** This is an optional lab to let you further explore using Generics. One suggested lab is to modify the Mixer so that it use a “custom” data structure to hold the word and its frequency count. This custom data structure could be contained in a List parameterized to your type.
- 🕒 **Duration:** 30 minutes

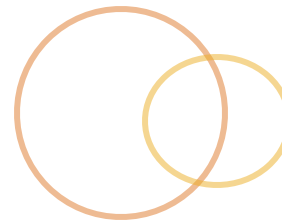
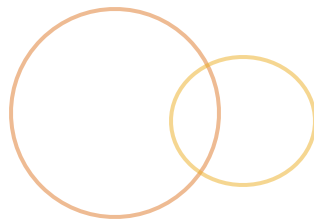
Metadata



Notes on Annotations



MetaData



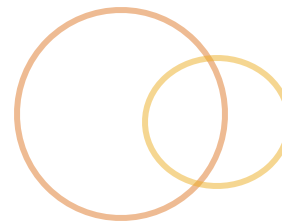
What is it?

- Typically described as “data about data”
- Usually provides additional information about data
- Basic example - comments in code
- More complex example - schema

Why is it needed?

- Provide additional data about data, outside of data
- Keeps data clean
- Can be used by tools to “learn” about the data, without interrogating data

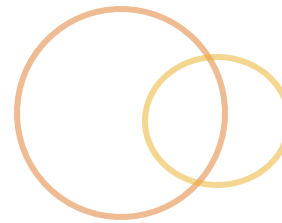
Annotations



What are they?

- Metadata facility for Java
 - Allowing you to provide additional data alongside Java classes
 - Similar to Javadoc “metadata” facility
- Expanded and formalized mechanism
 - “Competes” with Doclet / XDoclet
- Recognized by Java compiler and other tools

Annotations



Why do we need them?

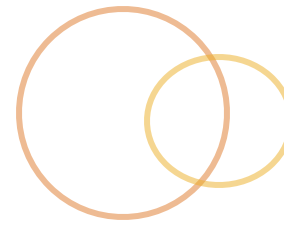
Additional data can be read:

- By the Compiler
- By source-code generation tools
- At run-time

Additional data can be used to:

- Generate boiler-plate code
- Maintain side-file dependencies
- Mark things for tracking purposes (like TODOs)

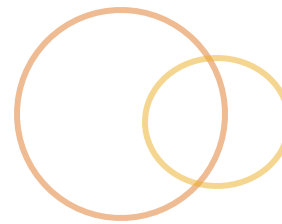
Annotations [cont.]



⦿ How do they work?

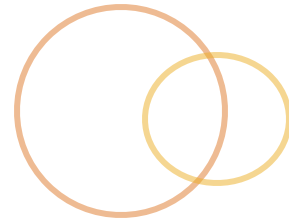
- ⦿ Don't affect program semantics; aren't allowed to disrupt execution
- ⦿ Represented as a new type within language
- ⦿ Have similar syntax to Javadoc
- ⦿ Applied like modifiers
- ⦿ Have constrained lifespan
- ⦿ Detected and interpreted by compiler

Annotation Type



- New type within language
 - `java.lang.annotation.Annotation`
 - Type can be annotated with other annotations
- Type like an interface
 - Use `@interface` instead of `interface`
 - Support methods
 - Must be declared without arguments
 - Methods can not throw `Exceptions`
 - Support name-value-pairs (NVP)
 - Can not have members; members defined through coding convention
 - Method name + return type define member as NVP
 - NVP can have default values (making it optional)

Annotation Syntax



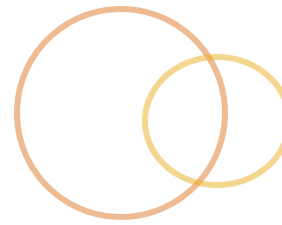
◎ Syntax similar to Javadoc syntax

- ◎ `@Deprecated` v. `@deprecated`
- ◎ `@` - represents annotation
- ◎ `Deprecated` - represents annotation type

◎ Syntax more robust than Javadoc syntax

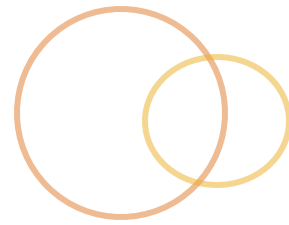
- ◎ Can pass NVP
 - ◎ `@SupressWarnings` - no NVP passed
 - ◎ `@SuppressWarnings(value={"unchecked", "fallthrough"})` - NVP passed
 - ◎ `@SuppressWarnings({"unchecked", "fallthrough"})` - NVP passed; short-hand
- ◎ Not white-space sensitive

Annotation Example



```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    /**
     * The set of warnings that are to be suppressed by the compiler in the
     * annotated element. Duplicate names are permitted. The second and
     * successive occurrences of a name are ignored. The presence of
     * unrecognized warning names is not an error: Compilers must
     * ignore any warning names they do not recognize. They are, however,
     * free to emit a warning if an annotation contains an unrecognized
     * warning name.
     *
     * <p>Compiler vendors should document the warning names they support in
     * conjunction with this annotation type. They are encouraged to cooperate
     * to ensure that the same names work across multiple compilers.
     */
    String[] value();
}
```

Provided Annotations



Two classifications:

Meta-annotations

- ◉ Annotate annotations

- ◉ Found in `java.lang.annotation`

- ◉ 4 main meta-annotations

- ◉ Used to define annotation behaviors

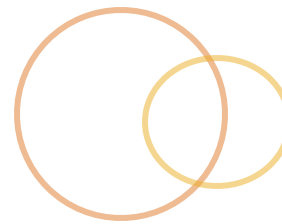
Annotations

- ◉ Core annotations

- ◉ Found in `java.lang`; automatically imported in source

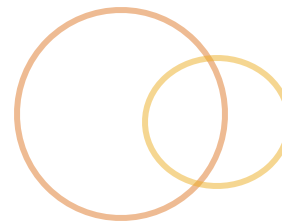
- ◉ 3 main annotations

Meta-Annotations



- Target
 - Identifies element applicability
 - Default / no value means applies to all elements
 - Possible values defined in `ElementType`
- Retention
 - Identifies lifespan of annotation
 - Three lifespans defined in `RetentionPolicy`:
 - `RetentionPolicy.SOURCE` - source only
 - `RetentionPolicy.CLASS` - source and class; not runtime
 - `RetentionPolicy.RUNTIME` - source, class, and runtime
 - Default / no value causes source only retention
- Documented - something that should be documented
- Inherited - annotation should be carried through inheritance

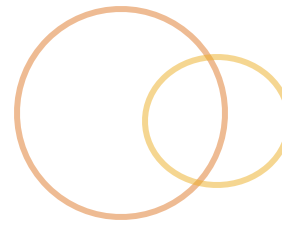
Core Annotations



◎ @Override

- ◎ Used to notify compiler that method is overridden representation of inherited method
 - ◎ Causes compiler to validate overridden signature
 - ◎ Generates compiler errors if not in sync
- ◎ @Target (ElementType.METHOD)
- ◎ @Retention (RetentionPolicy.SOURCE)

@Override Example



```
1 package examples.metadata;
2
3 + /**...*/
7 public class OverrideExample {
8     private String myValue;
9
10    @Override
11    public String toString() {
12        return myValue;
13    }
14 }
15
```

```
> javac OverrideExample.java
OverrideExample.java:10: method does not override a method from its superclass
    @Override
      ^
1 error
> 
```


Core Annotations [cont.]



◎ @Deprecated

- ◎ Marker annotation similar to @deprecated in Javadoc
- ◎ Used to notify compiler that use of @Deprecated element is discouraged
- ◎ No @Target specified
- ◎ `@Retention(RetentionPolicy.RUNTIME)`

Core Annotations [cont.]



◎ @SuppressWarnings

- ◎ Used to selectively turn off compiler warnings
- ◎ Code-level alternative to `-Xlint` compiler flag
- ◎ No Enum defining which warnings can be selected
- ◎ Works in “hierarchical” manner
- ◎ `@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})`
- ◎ `@Retention(RetentionPolicy.SOURCE)`

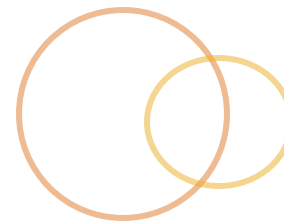
@SuppressWarnings Example



```
Advanced Java
> javac -Xlint SupressWarningsExample.java
SupressWarningsExample.java:15: warning: [unchecked] unchecked call to add(E) as a member of the raw type
java.util.List
    intList.add(1);
              ^
1 warning
> javac -Xlint SupressWarningsExample.java
> 
```

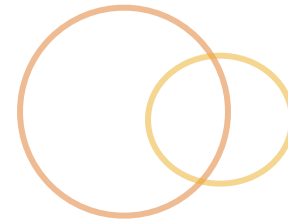
```
1 package examples.metadata;
2
3 import ...
4
5 /** ... */
6
11 public class SupressWarningsExample {
12
13     public List buildList() {
14         List intList = new ArrayList();
15         intList.add(1);
16         return intList;
17     }
18
19 }
20
```

```
1 package examples.metadata;
2
3 import ...
4
5 /** ... */
6
11 public class SupressWarningsExample {
12
13     @SuppressWarnings({"unchecked"})
14     public List buildList() {
15         List intList = new ArrayList();
16         intList.add(1);
17         return intList;
18     }
19
20 }
21
```



- 🕒 Description: Add a `toString` method to the `Month` enumeration. The `toString` method should return a title-case version of the month name (December). Apply the `@Override` annotations to `toString` method. `toString` is not a valid method to override. You should see compiler errors. Clear up any compile errors by changing `toString` to `toString`.
- 🕒 Duration: 15 minutes

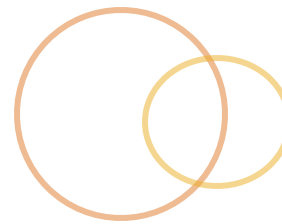
Covariant Returns



Simplifying Type-safe Returns



Covariant Returns



What are they?

- Mechanism added to language
- Allowing return type of inherited method to be narrowed
- Applies to method over-riding *not* over-loading

Why do they exist?

- Needed to support generics mechanism
- Removes narrowing cast on polymorphic returns
- Prevents run-time `ClassCastException`s on returns
- Provides compile-time type dependency checking

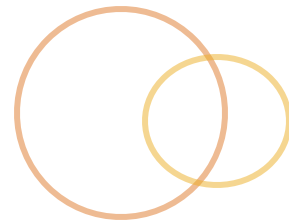
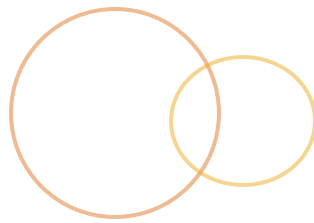
Covariant Return Example



```
1 package examples.covariantreturns;
2
3 /**...*/
10 public class Parent {
11
12     private String name;
13     private String value;
14
15     public Object getName() {
16         return name;
17     }
18
19     public Object getValue() {
20         return value;
21     }
22 }
23
```

```
1 package examples.covariantreturns;
2
3 /**...*/
10 public class Child extends Parent {
11
12     @Override
13     public String getName() {
14         return (String) super.getName();
15     }
16
17     @Override
18     public String getValue() {
19         return (String) super.getValue();
20     }
21 }
22
```

Summary



Five advanced language enhancements

- ◎ **Enums** - type supporting Enumeration Pattern
- ◎ **Generics** - mechanism for creating parameterized types and methods
- ◎ **Annotations** - mechanism to define additional information without effecting execution
- ◎ **Covariant returns** - mechanism to narrow return type