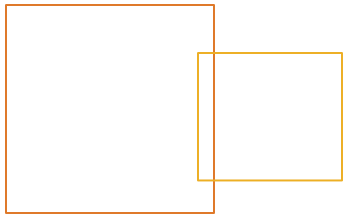
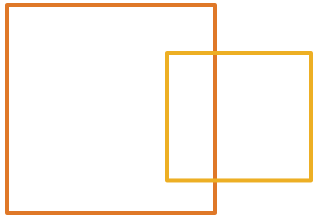
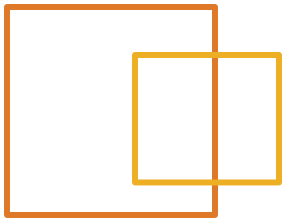
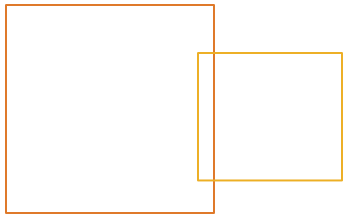
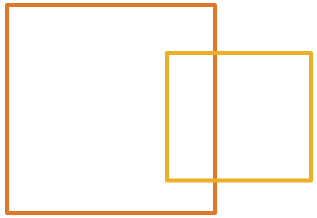
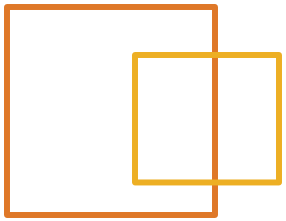





Fast Track to Java

Customized for Starbucks

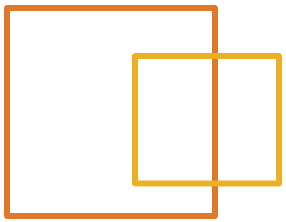
Delivered by DevelopIntelligence


Enums and Annotations

Advanced Language Features

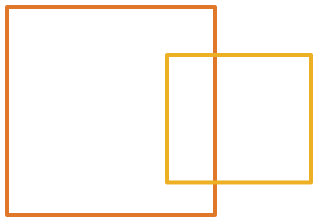
Presentation Topics

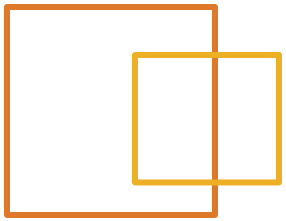


In this section, we will cover:

- **Enums** - type supporting Enumeration Pattern
- **Annotations** - mechanism to define additional information without effecting execution
- **Covariant returns** - mechanism to narrow return type

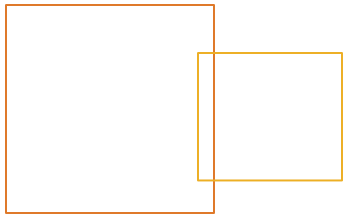
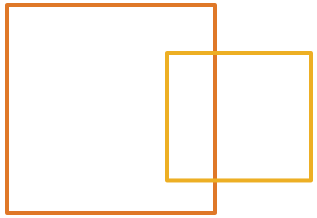
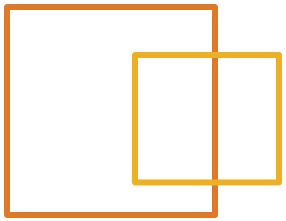
Objectives





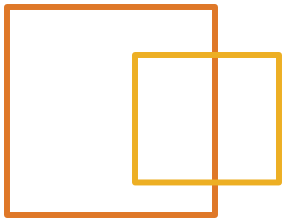
At the end of this module you should be able to:

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

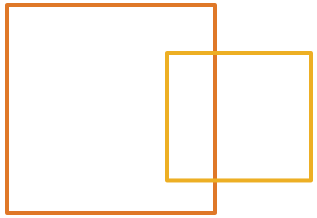

Type-Safe Enumerations

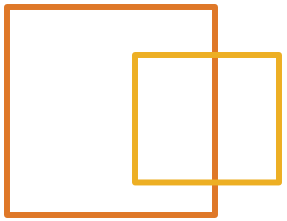
Type-Safe Enumerations



- 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

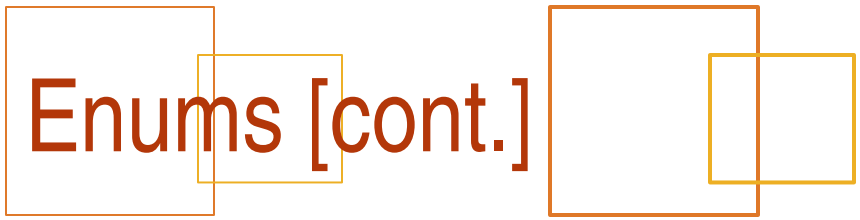
Enums

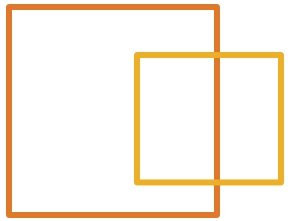




- Why do they exist?
  - Historically implemented using an enum pattern
  - Common problems with enum pattern:
    - Not type-safe
    - No separate namespace (values typically defined as fields)
    - Based on primitive values that may change
  - Laborious to develop using enum pattern
    - Creates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

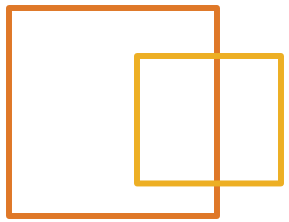
Enums [cont.]





- 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 are types
  - 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
```

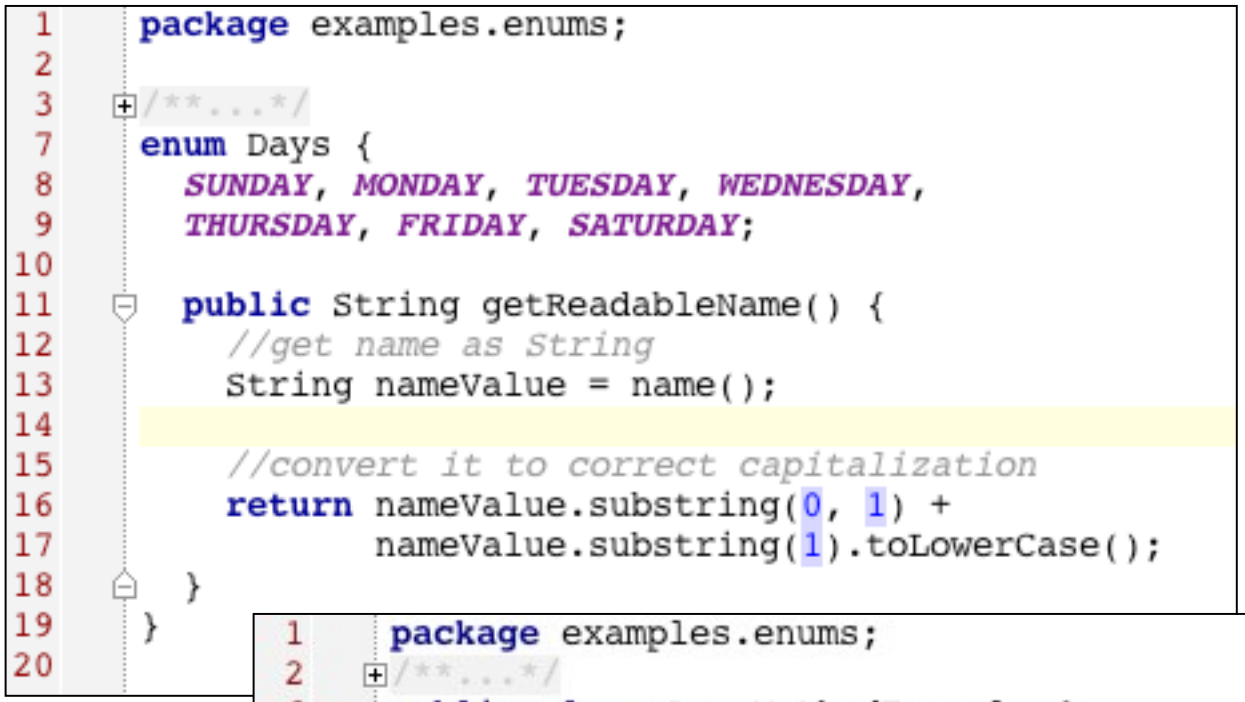
Prints:

```
SUNDAY
MONDAY
TUESDAY
WEDNESDAY
THURSDAY
FRIDAY
SATURDAY
```

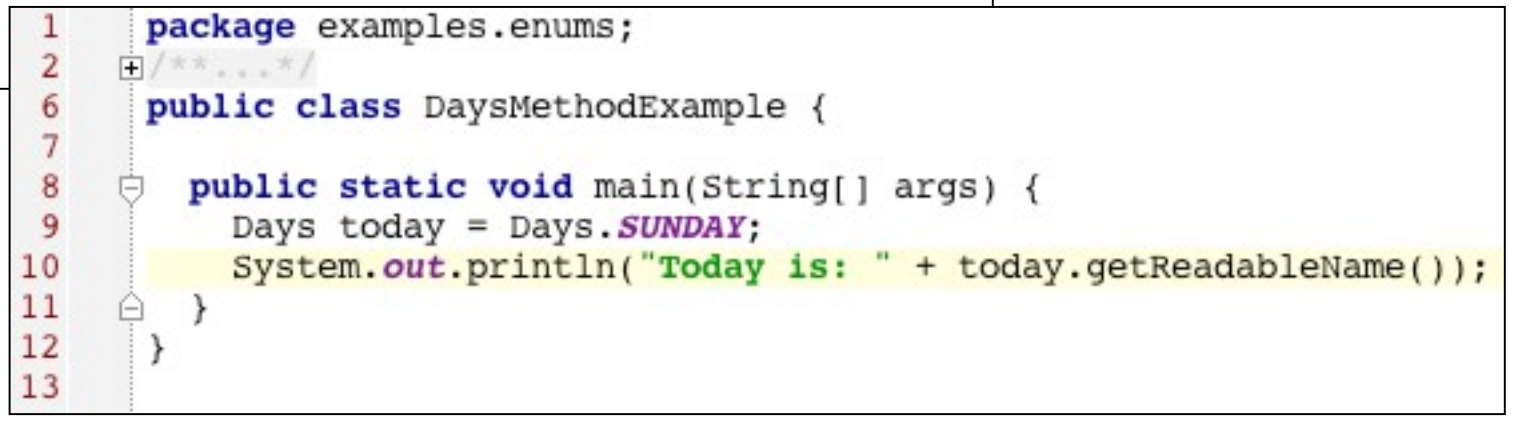
Working with an Enum [cont.]

- Enums are types
  - Support 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  /**...*/
7  enum Days {
8    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
9    THURSDAY, FRIDAY, SATURDAY;
10
11   public String getReadableName() {
12     //get name as String
13     String nameValue = name();
14
15     //convert it to correct capitalization
16     return nameValue.substring(0, 1) +
17            nameValue.substring(1).toLowerCase();
18   }
19 }
20
```



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

Working With an Enum [cont.]

- Enums are types
  - Methods can be over-ridden
    - Enum-defined methods
    - Object methods
  - Method overriding 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.]

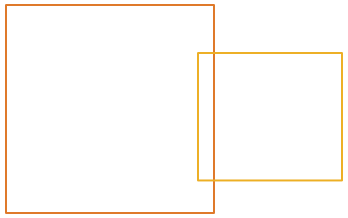
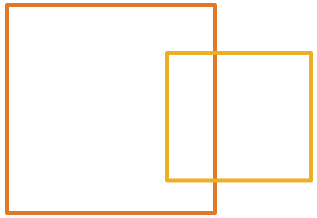
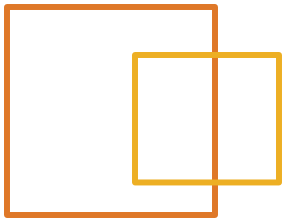
- Enums are types
  - Enums support constructors
    - Constructors are private
    - Used to initialize instance variables
    - Provide type-safe instance creation

Enum Constructor Example

```
1  package examples.enums;
2
3  /**...*/
7  enum DaysToo {
8    SUNDAY ("Sunday"), MONDAY ("Monday"),
9    TUESDAY ("Tuesday"), WEDNESDAY ("Wednesday"),
10   THURSDAY ("Thursday"), FRIDAY ("Friday"),
11   SATURDAY;
12
13   private String readableName;
14
15   DaysToo() {
16     readableName = name();
17   }
18
19   DaysToo(String s) {
20     readableName = s;
21   }
22
23   public String getReadableName() {
24     return readableName;
25   }
26
27   public String toString() {
28     return getReadableName();
29   }
30 }
31
```

Advanced Enum Features

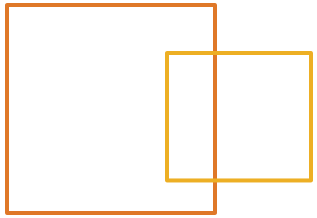
- Enums are types
  - No enum - enum inheritence chains
  - No class - enum inheritence chains
  - Can implement interfaces
- Two new enumeration oriented collections
  - `EnumMap`– converts enum fields into map keys
  - `EnumSet`– converts enum fields into a set

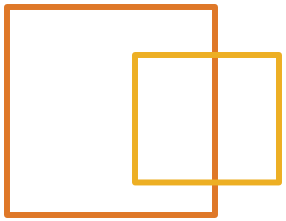





Metadata

Notes on Annotations

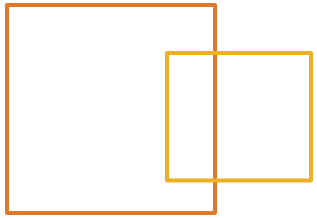
MetaData

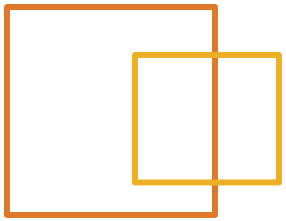




- What is it?
  - Typically described as “data about data”
  - Usually provides additional information about data
  - Basic example - comments in code
  - More complex example - schema
- Why is it needed?
  - Provide additional data about data, outside of data
  - Keeps data clean
  - Can be used by tools to “learn” about the data, without interrogating data

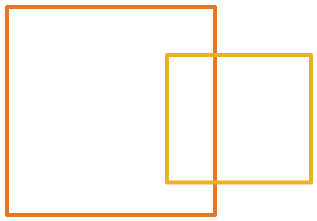
Annotations

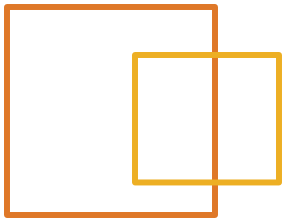




- What are they?
  - Metadata facility for Java
    - Allowing you to provide additional data alongside Java classes
    - Similar to Javadoc “metadata” facility
  - Expanded and formalized mechanism
    - “Competes” with Doclet / XDoclet
  - Recognized by Java compiler and other tools

Annotations



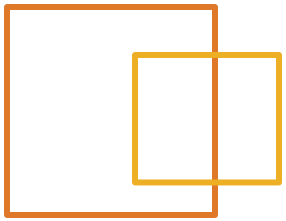


- Why do we need them?
  - Additional data can be read:
    - By the Compiler
    - By source-code generation tools
    - At run-time
  - Additional data can be used to:
    - Generate boiler-plate code
    - Maintain side-file dependencies
    - Mark things for tracking purposes (like TODOs)

Annotations [cont.]

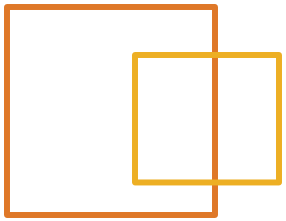
- How do they work?
  - Don’t affect program semantics; aren’t allowed to disrupt execution
  - Represented as a new type within language
  - Have similar syntax to Javadoc
  - Applied like modifiers
  - Have constrained lifespan
  - Detected and interpreted by compiler

Annotation Type



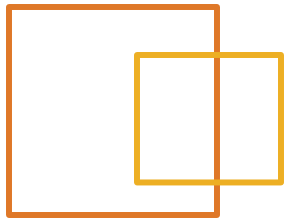
- New type within language
  - `java.lang.annotation.Annotation`
  - Type can be annotated with other annotations
- Type like an interface
  - Use `@interface` instead of interface
  - Support methods
    - Must be declared without arguments
    - Methods can not throw `Exceptions`
  - Support name-value-pairs (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



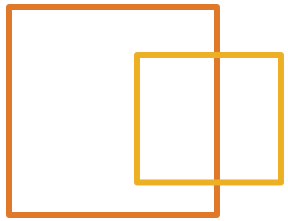
- Syntax similar to Javadoc syntax
  - `@Deprecated` v. `@deprecated`
    - `@` - represents annotation
    - `Deprecated` - represents annotation type
- Syntax more robust than Javadoc syntax
  - Can pass NVP
    - `@SupressWarnings` - no NVP passed
    - `@SuppressWarnings(value={“unchecked, fallthrough”})` - 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 <i>not</i> 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

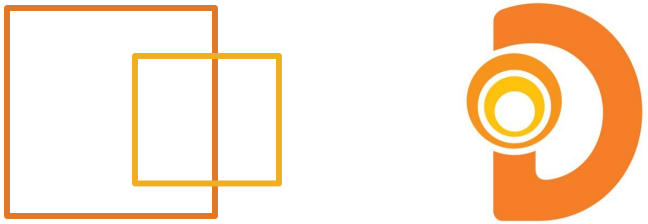


- Two classifications:
  - Meta-annotations
    - Annotate annotations
    - Found in `java.lang.annotation`
    - 4 main meta-annotations
    - Used to define annotation behaviors
  - Annotations
    - Core annotations
    - Found in `java.lang`; automatically imported in source
    - 3 main annotations

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 inheritence

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

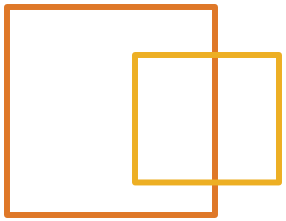
```
1   package examples.metadata;
2
3   /**...*/
7   public class OverrideExample {
8     private String myValue;
9
10    @Override
11    public String tostring() {
12      return myValue;
13    }
14  }
15
```

```
 > 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.]



- `@SupressWarnings`
  - 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)`

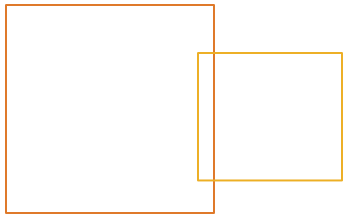
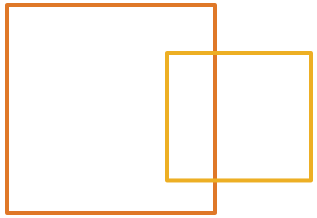
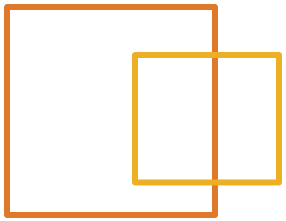
@SuppressWarnings Example

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 ...
 5
 6  /**...*/
11  public class SupressWarningsExam
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 ...
 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
```

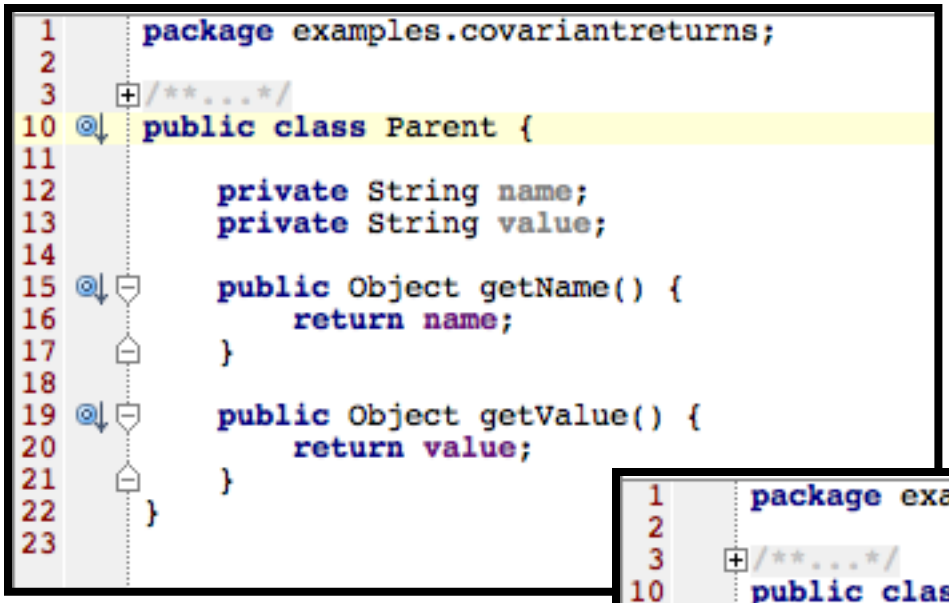

Covariant Returns

Simplifying Type-safe Returns

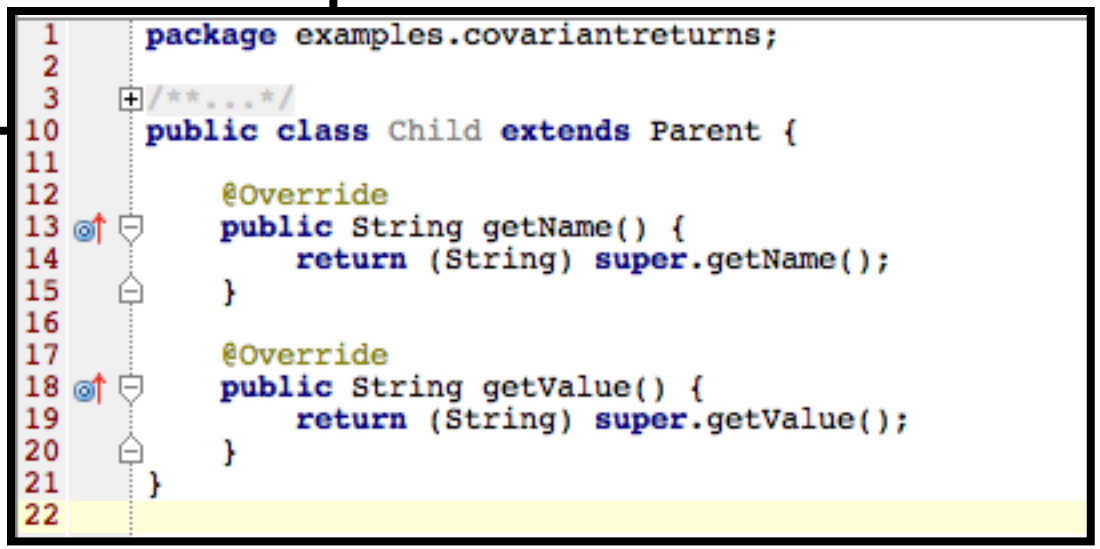
Covariant Returns

- 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 `ClassCastExceptions` on returns
  - Provides compile-time type dependency checking

Covariant Return Example

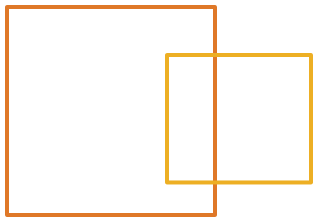


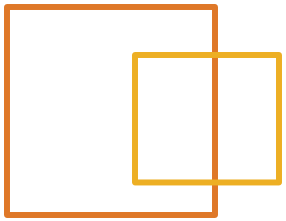
```
1   package examples.covariantreturns;
2
3   /**...*/
10  public class Parent {
11
12      private String name;
13      private String value;
14
15      public Object getName() {
16          return name;
17      }
18
19      public Object getValue() {
20          return value;
21      }
22  }
23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



```
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





In this module, we covered:

- **Enums** - type supporting Enumeration Pattern
- **Annotations** - mechanism to define additional information without effecting execution
- **Covariant returns** - mechanism to narrow return type