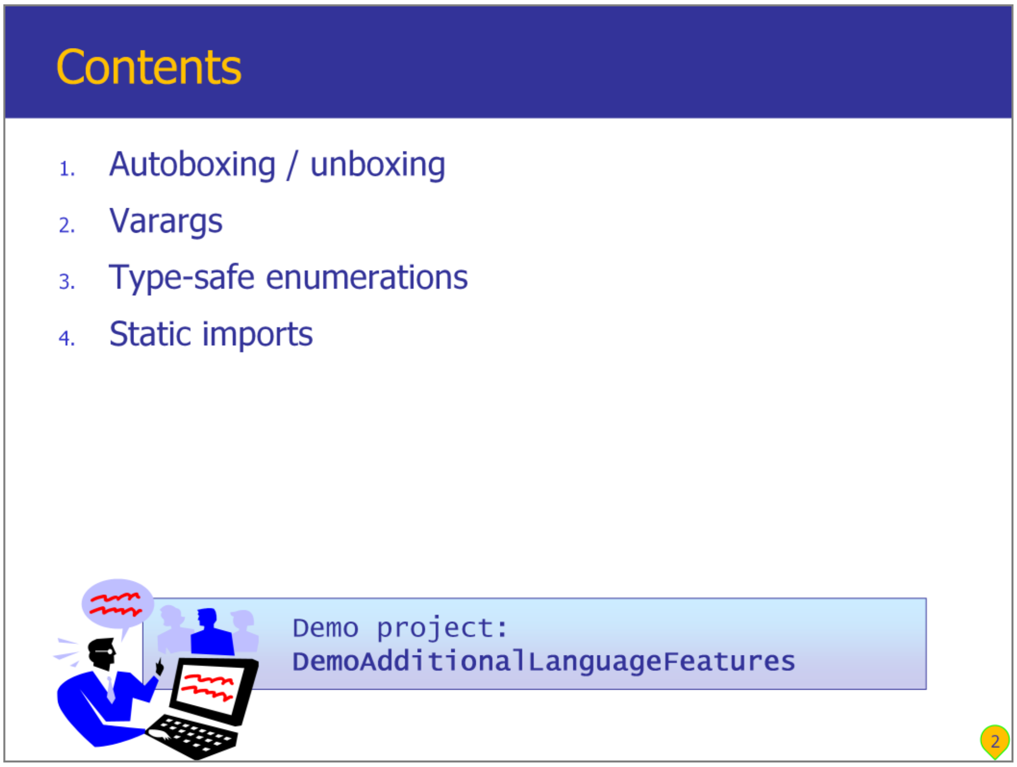
Additional Java Language Features

This chapter describes miscellaneous language features that have been added to Java in recent years, to rectify language deficiencies and omissions in earlier versions of the language.

We've grouped these miscellaneous features into this chapter, because they don't really fit in anywhere else ☺.



Contents

1. Autoboxing / unboxing
2. Varargs
3. Type-safe enumerations
4. Static imports

Demo project:
`DemoAdditionalLanguageFeatures`

2

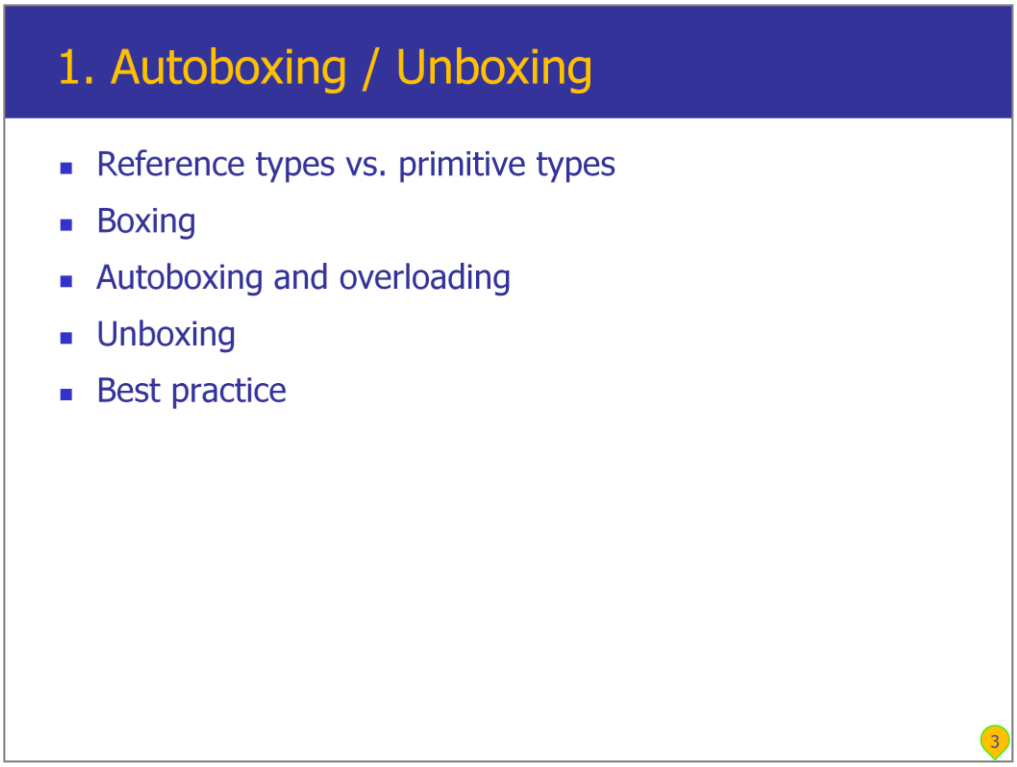
Section 1 introduces the concept of boxing and unboxing, and explains how the rules have been relaxed in Java to allow automatic boxing and unboxing. This is important, even if it might sound a bit cryptic at this stage!

Section 2 describes how to define methods that take a variable number of arguments. A good example is the `System.out.printf()` method, which allows you to pass any number of parameters along with your format string.

Section 3 shows how to define enums in Java. Conceptually, an enum is like an integer that has a fixed set of allowable values. This isn't quite how enums work in Java, but it's a good mental image for the moment.

Section 4 explains how to use static imports. This makes your code cleaner if you need to use lots of `static` members in a class.

The demos for chapter are located in the `DemoAdditionalLanguageFeatures` project.



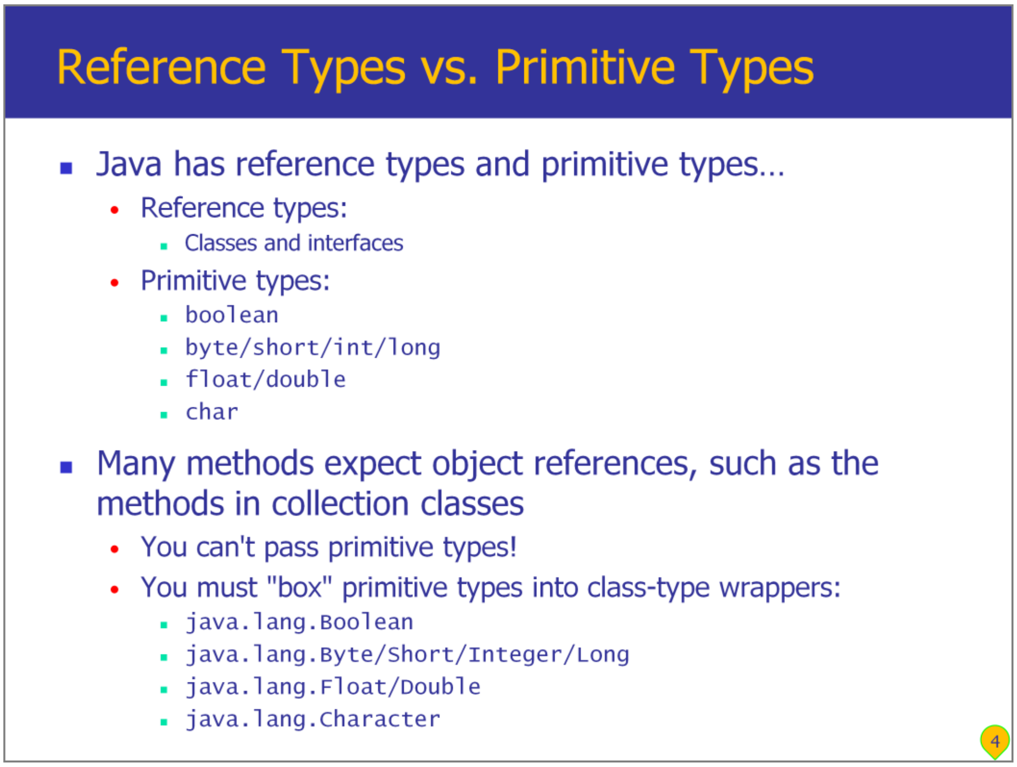
1. Autoboxing / Unboxing

- Reference types vs. primitive types
- Boxing
- Autoboxing and overloading
- Unboxing
- Best practice

3

In this section, we describe what boxing and unboxing are, and then we explain how the rules for boxing and unboxing have been relaxed in the Java language.

In a nutshell, boxing and unboxing refer to the mechanism whereby primitive values (such as `int`) are converted to and from wrapper class objetcs (such as `Integer`).



Reference Types vs. Primitive Types

- Java has reference types and primitive types…
  - Reference types:
    - Classes and interfaces
  - Primitive types:
    - `boolean`
    - `byte/short/int/long`
    - `float/double`
    - `char`
- Many methods expect object references, such as the methods in collection classes
  - You can't pass primitive types!
  - You must "box" primitive types into class-type wrappers:
    - `java.lang.Boolean`
    - `java.lang.Byte/Short/Integer/Long`
    - `java.lang.Float/Double`
    - `java.lang.Character`

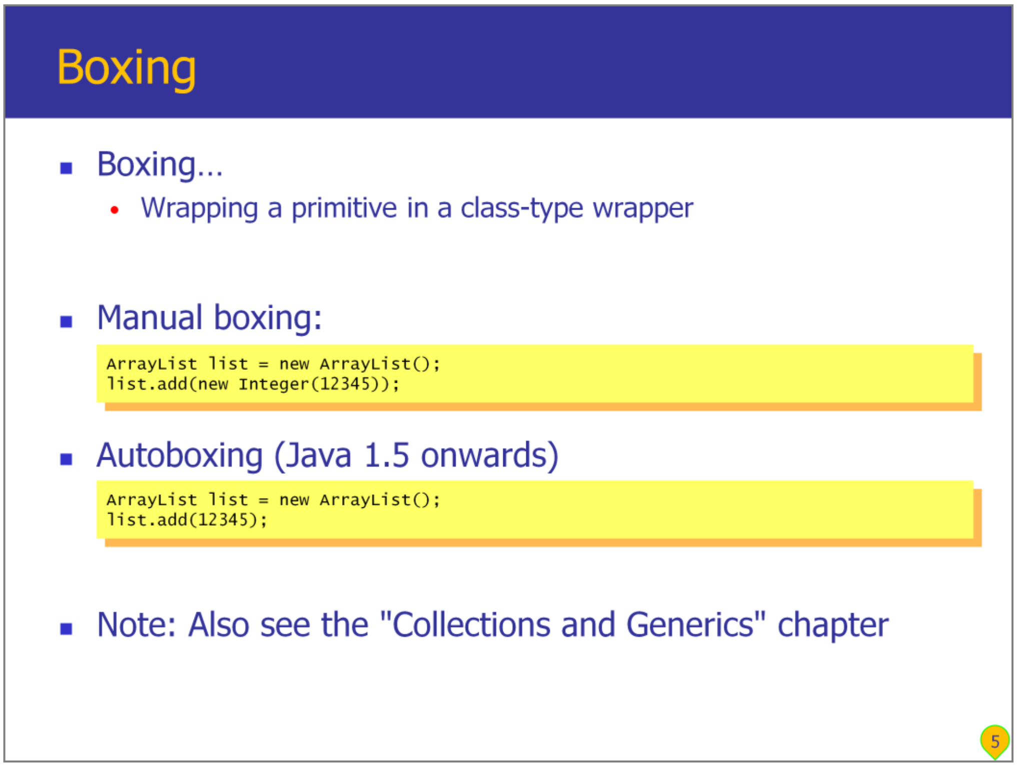
4

As we pointed out at the start of the course, Java differentiates between primitive types and everything else. Anything that isn't a primitive type is basically a class type (or interface) that inherits from the `Object` class.

This is relevant because there are many methods in Java that require you to pass an `Object` as a parameter. Via the magic of polymorphism (discussed later in the course), this means you can pass any kind of object (i.e. any class/interface type), because all classes/interfaces inherit from `Object`. However, you can't pass primitive-type values, because primitive types are not classes, and therefore do not inherit from `Object`.

This is where wrapper classes enter the equation. Wrapper classes really are classes, and as such they inherit from `Object`. Therefore, you're allowed to pass wrapper-class objects into any method that expects an `Object` parameter.

The trick, then, is to convert a primitive-type value into a corresponding wrapper-class object, and then pass the wrapper-class object into the method. This process is known as "boxing". You can imagine the wrapper-class object being a box that contains the primitive-type value inside it.



Boxing

- Boxing...
  - Wrapping a primitive in a class-type wrapper
- Manual boxing:

```
ArrayList list = new ArrayList();
list.add(new Integer(12345));
```

- Autoboxing (Java 1.5 onwards)

```
ArrayList list = new ArrayList();
list.add(12345);
```

- Note: Also see the "Collections and Generics" chapter

5

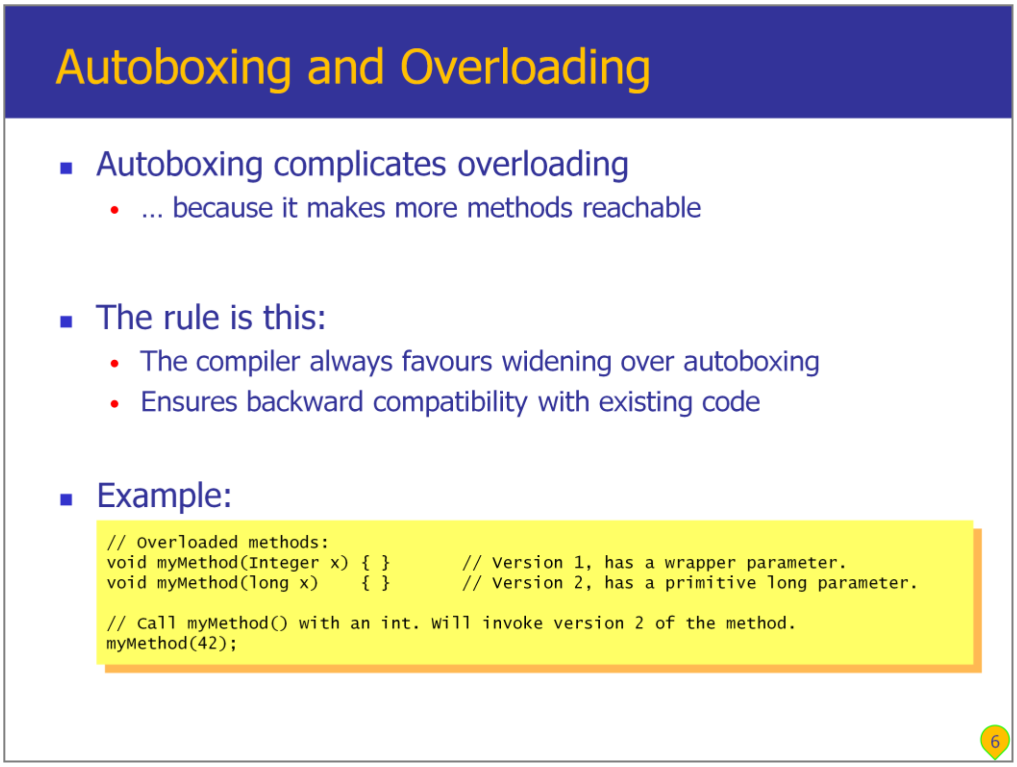
As we identified on the previous slide, "boxing" is the mechanism whereby you create a wrapper-class object to hold a copy of a primitive-type value. The main use of boxing is to pass primitive values into methods that expect an `Object` parameter.

Prior to Java 1.5, boxing was a purely manual affair. You had to create the wrapper-class object yourself, and initialize it with your primitive value. The first code block in the slide shows this rather messy approach.

From Java 1.5 onwards, boxing is now automatic. You can seemingly pass a primitive-type value directly into a method that expects an `Object` parameter. The second code block in the slide shows this improved approach.

Don't be fooled by autoboxing, however. A wrapper-class object is still being created under the covers; it's just that the compiler is doing it implicitly, to make your code a bit easier to write.

Also note that the use of generics greatly reduces the need for boxing and unboxing. We'll discuss generics later in the course.



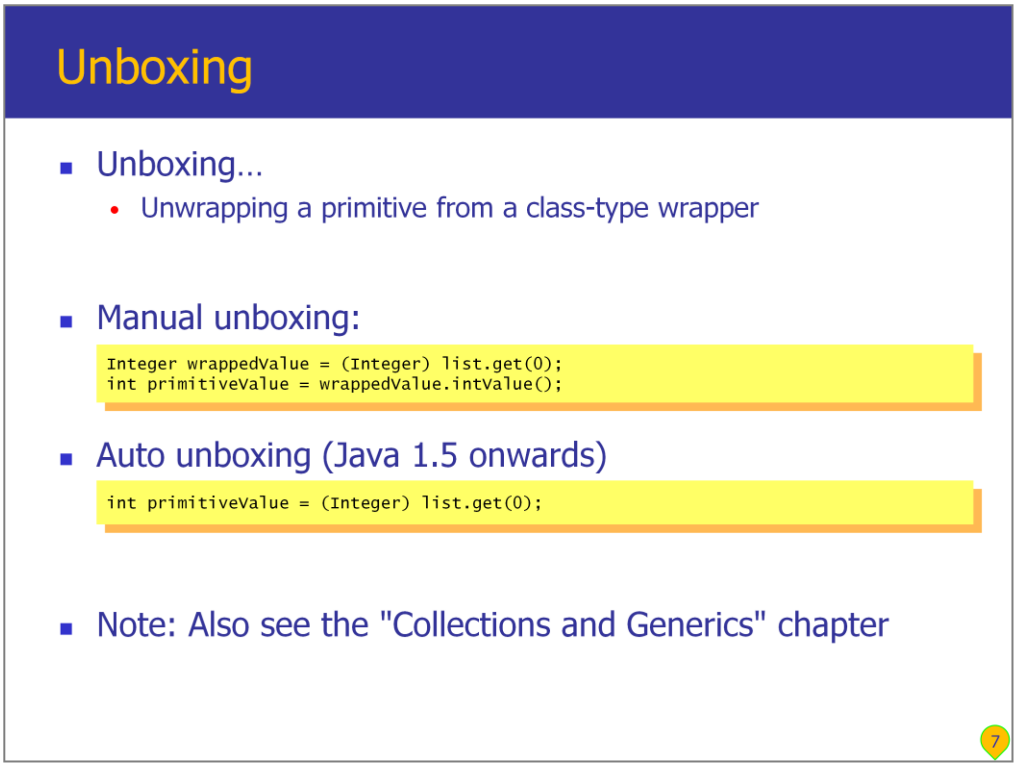
Autoboxing and Overloading

- Autoboxing complicates overloading
  - … because it makes more methods reachable
- The rule is this:
  - The compiler always favours widening over autoboxing
  - Ensures backward compatibility with existing code
- Example:

```
// Overloaded methods:
void myMethod(Integer x) { }       // Version 1, has a wrapper parameter.
void myMethod(long x)    { }       // Version 2, has a primitive long parameter.

// Call myMethod() with an int. Will invoke version 2 of the method.
myMethod(42);
```

6

This slide outlines some technicalities if you choose to define overloaded methods that take various combinations of primitive-type and wrapper-class-type parameters. The rules are rather arcane, so you're generally best off avoiding such overloading altogether. Seriously!



Unboxing

- Unboxing…
  - Unwrapping a primitive from a class-type wrapper
- Manual unboxing:

```
Integer wrappedValue = (Integer) list.get(0);
int primitiveValue = wrappedValue.intValue();
```

- Auto unboxing (Java 1.5 onwards)

```
int primitiveValue = (Integer) list.get(0);
```

- Note: Also see the "Collections and Generics" chapter

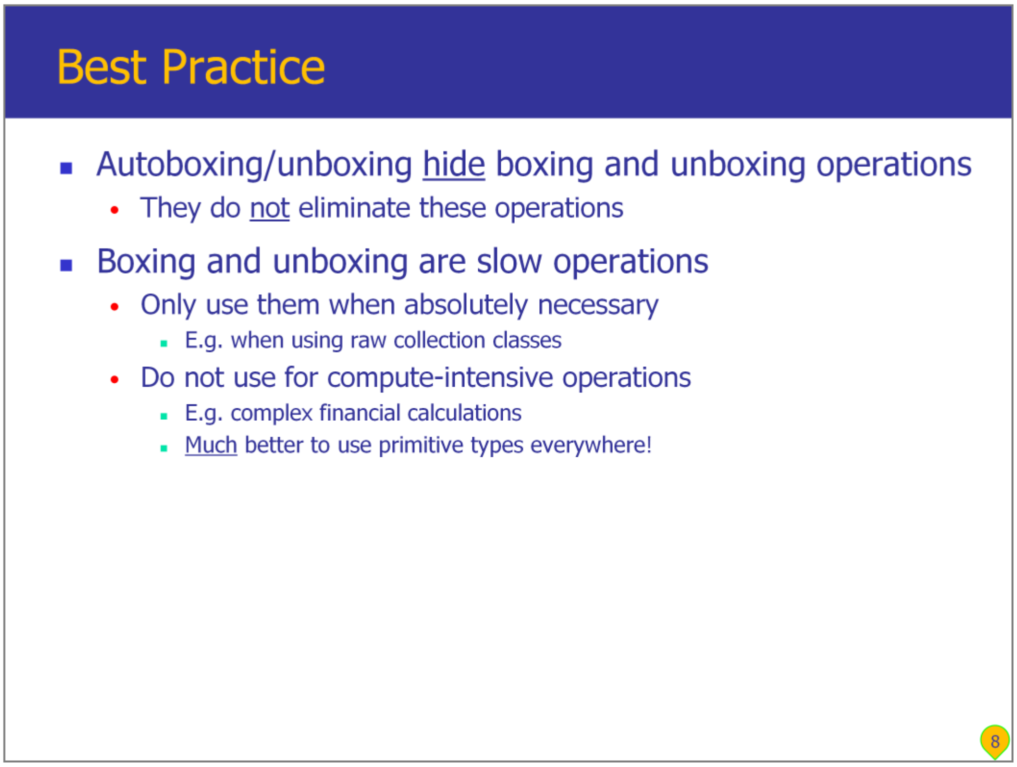
7

Unboxing is the opposite of boxing. Unboxing occurs where you have a wrapper-class object and you extract the primitive-type value that it contains.

As with boxing, unboxing was manual prior to Java 1.5. The first code block in the slide shows an example of manual unboxing. Note the following points:

- The first statement gets an item from a list. The `get()` method return type is `Object`, so we convert it into `Integer` (we're assuming the list contains `Integer` objects in this example).
- The second statement manually unboxes the primitive-type value (i.e. an `int`) from the wrapper-class object (i.e. the `Integer` object).

From Java 1.5 onwards, unboxing is now automatic. You can seemingly assign a wrapper-class object directly to a primitive-type variable. The second code block in the slide shows this improved approach. Bear in mind that unboxing is still actually happening under the covers; it's just that the syntax is cleaner.

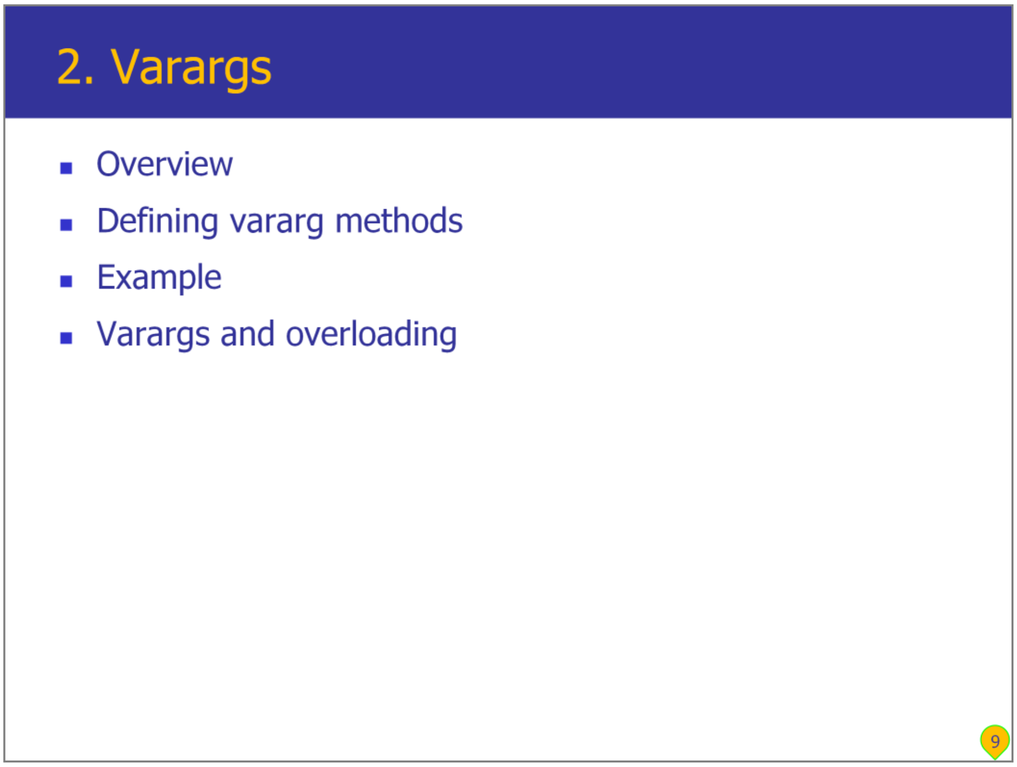


Best Practice

- Autoboxing/unboxing hide boxing and unboxing operations
  - They do not eliminate these operations
- Boxing and unboxing are slow operations
  - Only use them when absolutely necessary
    - E.g. when using raw collection classes
  - Do not use for compute-intensive operations
    - E.g. complex financial calculations
    - Much better to use primitive types everywhere!

8

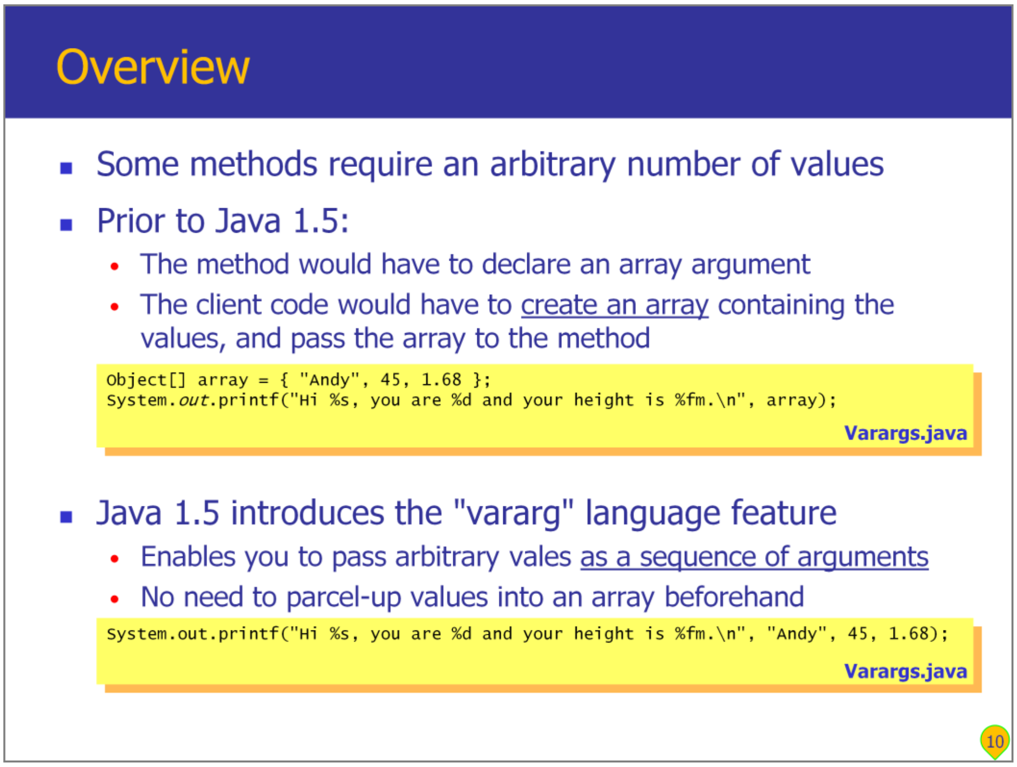
Here are some guidelines of how to use boxing and unboxing in your code. The general recommendation is for you to be aware of where boxing and unboxing is taking place, and to try to avoid it where possible.



2. Varargs

- Overview
- Defining vararg methods
- Example
- Varargs and overloading

This section describes how to define methods that take a variable number of arguments. There aren't too many methods that require this flexibility, but we've already seen some examples of where this feature comes in handy in the Java SE library. Now we're going to show how to write variable-argument methods for ourselves.



Overview

- Some methods require an arbitrary number of values
- Prior to Java 1.5:
  - The method would have to declare an array argument
  - The client code would have to create an array containing the values, and pass the array to the method

```
Object[] array = { "Andy", 45, 1.68 };
System.out.printf("Hi %s, you are %d and your height is %fm.\n", array);
```

Varargs.java

- Java 1.5 introduces the "vararg" language feature
  - Enables you to pass arbitrary vales as a sequence of arguments
  - No need to parcel-up values into an array beforehand

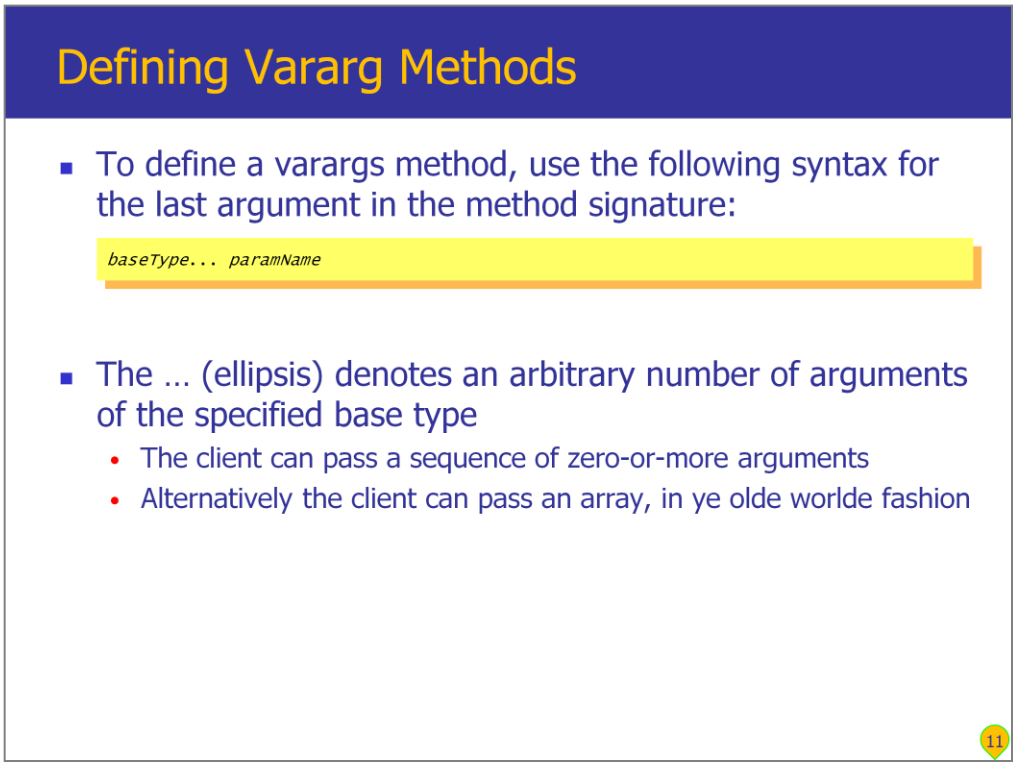
```
System.out.printf("Hi %s, you are %d and your height is %fm.\n", "Andy", 45, 1.68);
```

Varargs.java

10

Prior to Java 1.5, the only way to define a method that was flexible enough to take a variable number of arguments was to declare an array parameter. The client code could then pass in an array with as many elements as necessary. The first code block in the slide shows this approach. This is all very well and good, and the mechanism does work. The downside is that it forces the client code to bundle the values up into an array first.

Java 1.5 introduced the "varargs" language feature, which allows you to implement variable-argument methods properly. The client code can pass as many arguments as it likes, using regular argument-passing syntax. The second code block in the slide shows how client code will look now.



Defining Vararg Methods

- To define a varargs method, use the following syntax for the last argument in the method signature:

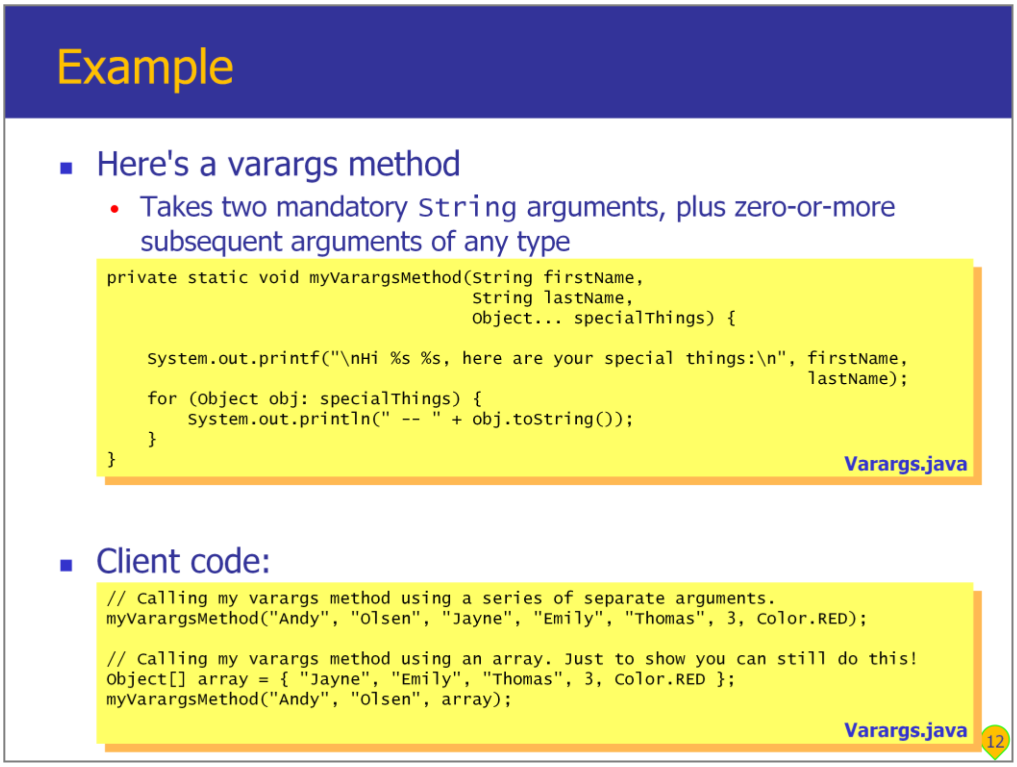
```
baseType... paramName
```

- The … (ellipsis) denotes an arbitrary number of arguments of the specified base type
  - The client can pass a sequence of zero-or-more arguments
  - Alternatively the client can pass an array, in ye olde worlde fashion

11

To define a method that can take a variable number of arguments, use the syntax shown above for the final parameter in the method signature. Note that the ellipsis (…) is real syntax here.

The client can pass as many argument values as it likes, using regular argument-passing syntax. Under the covers, the arguments are converted back into an array internally. Thus the recipient method uses normal array-processing syntax to access the incoming elements.



Example

- Here's a varargs method
  - Takes two mandatory `String` arguments, plus zero-or-more subsequent arguments of any type

```
private static void myVarargsMethod(String firstName,
                                    String lastName,
                                    Object... specialThings) {

    System.out.printf("\nHi %s %s, here are your special things:\n", firstName,
                                                                       lastName);
    for (Object obj: specialThings) {
        System.out.println(" -- " + obj.toString());
    }
}
```

Varargs.java

- Client code:

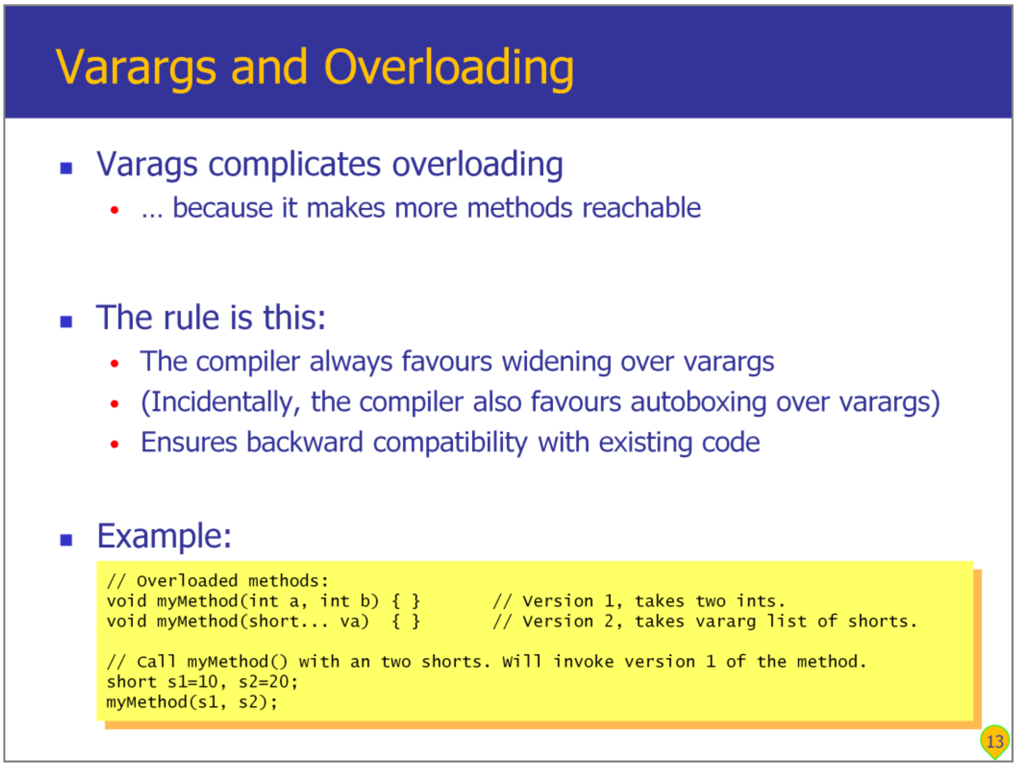
```
// Calling my varargs method using a series of separate arguments.
myVarargsMethod("Andy", "Olsen", "Jayne", "Emily", "Thomas", 3, Color.RED);

// Calling my varargs method using an array. Just to show you can still do this!
Object[] array = { "Jayne", "Emily", "Thomas", 3, Color.RED };
myVarargsMethod("Andy", "Olsen", array);
```

Varargs.java

12

This slide shows a complete example of how to define and invoke a variable-argument method. Note the following points:

- The first code block defines a method named `myVarargsMethod()`. The method takes two required `String` arguments, followed by any number of additional arguments. These additional arguments can be `Object` or any type that inherits from `Object`.
- The second code block shows how to invoke `myVarargsMethod()`, either by using regular argument-passing syntax (☺) or by using array syntax if you really want to (☹).



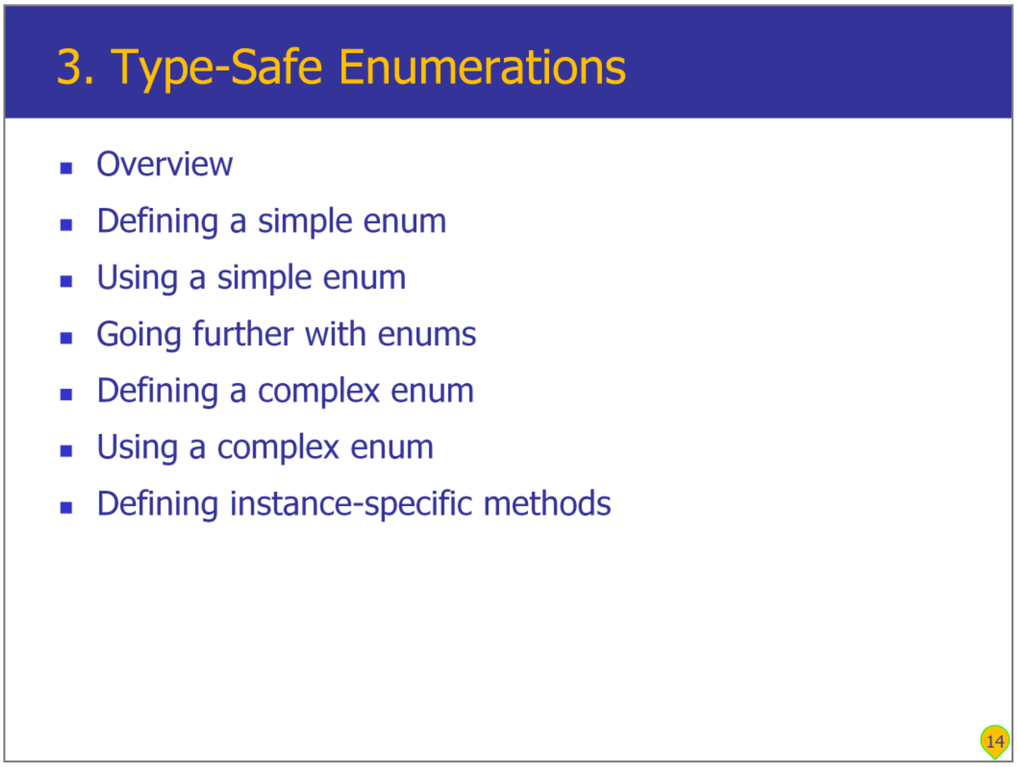
Varargs and Overloading

- Varags complicates overloading
  - … because it makes more methods reachable
- The rule is this:
  - The compiler always favours widening over varargs
  - (Incidentally, the compiler also favours autoboxing over varargs)
  - Ensures backward compatibility with existing code
- Example:

```
// Overloaded methods:
void myMethod(int a, int b) { }       // Version 1, takes two ints.
void myMethod(short... va)  { }       // Version 2, takes vararg list of shorts.

// Call myMethod() with an two shorts. Will invoke version 1 of the method.
short s1=10, s2=20;
myMethod(s1, s2);
```

13

Varargs can introduce some nasty complexities if you use it in conjunction with method overloading. This slide summarizes the rules, but to be honest you're best off avoiding mixing varargs with overloading altogether!



3. Type-Safe Enumerations

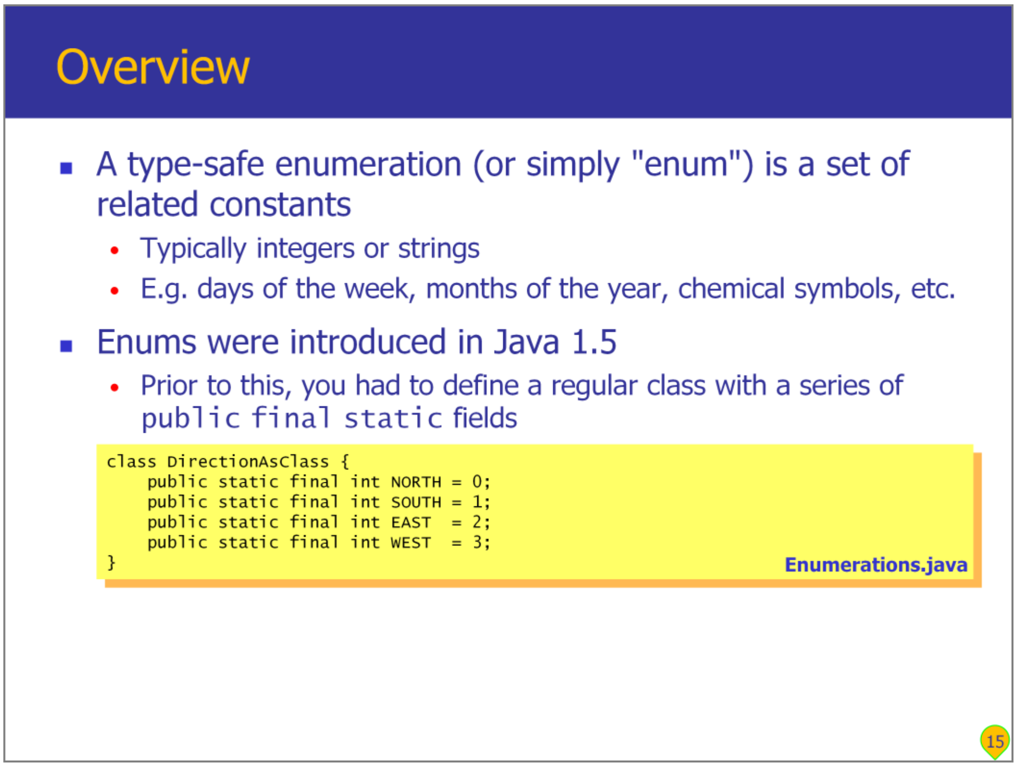
- Overview
- Defining a simple enum
- Using a simple enum
- Going further with enums
- Defining a complex enum
- Using a complex enum
- Defining instance-specific methods

14

Now it's time to turn our attention to enums, or "type-safe enumerations" to give them their official name in the Java specification.

An enum behaves similarly to an integer with a constrained set of allowable values. For example, you can image an enum type named `Direction` with named values such as `North`, `South`, `East`, and `West`.

That's the idea, but it's not how enums actually work internally in Java. We're going to explain the details in this section.



Overview

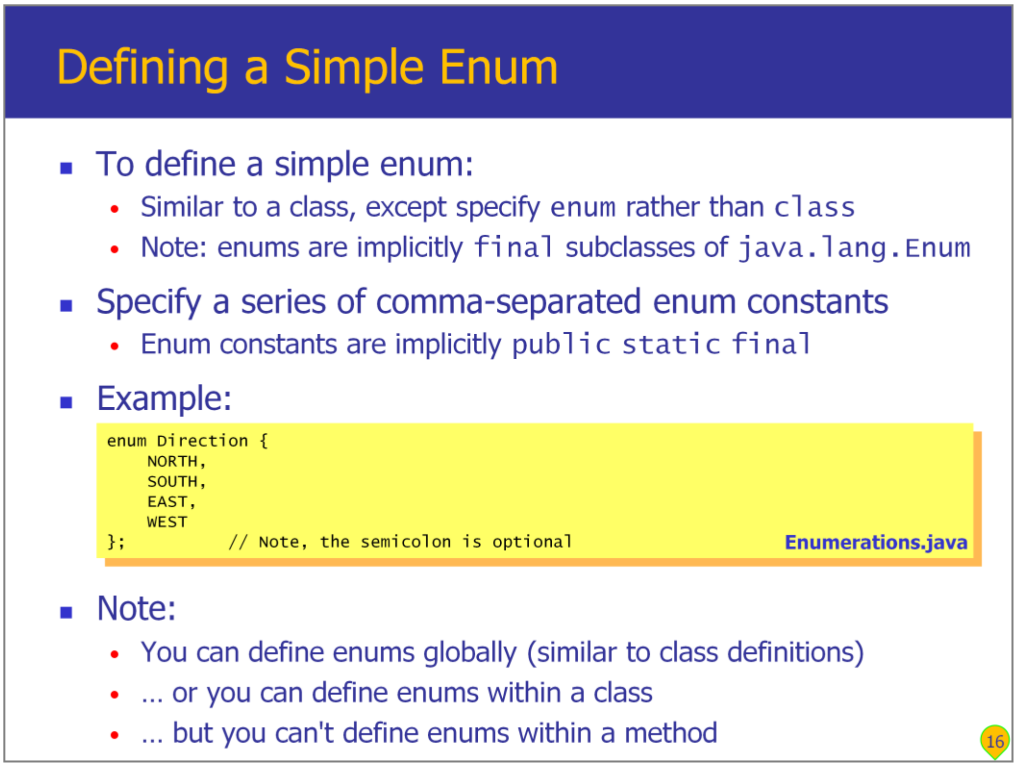
- A type-safe enumeration (or simply "enum") is a set of related constants
  - Typically integers or strings
  - E.g. days of the week, months of the year, chemical symbols, etc.
- Enums were introduced in Java 1.5
  - Prior to this, you had to define a regular class with a series of `public final static` fields

```
class DirectionAsClass {
    public static final int NORTH = 0;
    public static final int SOUTH = 1;
    public static final int EAST  = 2;
    public static final int WEST  = 3;
}
```

Enumerations.java

15

Enums were introduced in Java 1.5. Before that, if you wanted to define a type with a fixed set of allowable values, you had to define a class and specify the allowed values as `static final` fields in the class. The example in the slide shows this traditional approach.



Defining a Simple Enum

- To define a simple enum:
  - Similar to a class, except specify `enum` rather than `class`
  - Note: enums are implicitly `final` subclasses of `java.lang.Enum`
- Specify a series of comma-separated enum constants
  - Enum constants are implicitly `public static final`
- Example:

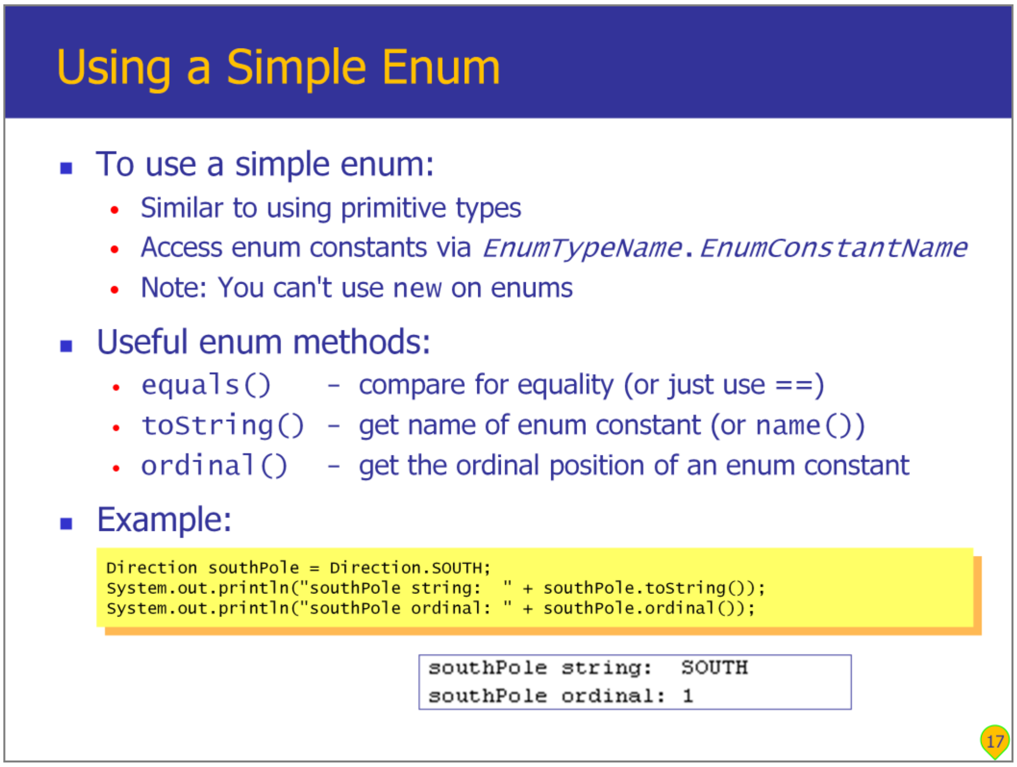
```
enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST
};          // Note, the semicolon is optional
```

Enumerations.java

- Note:
  - You can define enums globally (similar to class definitions)
  - ... or you can define enums within a class
  - ... but you can't define enums within a method

16

From Java 1.5 onwards, `enum` is a keyword in Java. You can now define enum types using the syntax shown in the slide. Typically, you embed such declarations within another class, so that the enum type is scoped within that class.



Using a Simple Enum

- To use a simple enum:
  - Similar to using primitive types
  - Access enum constants via *EnumTypeName.EnumConstantName*
  - Note: You can't use `new` on enums
- Useful enum methods:
  - `equals()` – compare for equality (or just use `==`)
  - `toString()` – get name of enum constant (or `name()`)
  - `ordinal()` – get the ordinal position of an enum constant
- Example:

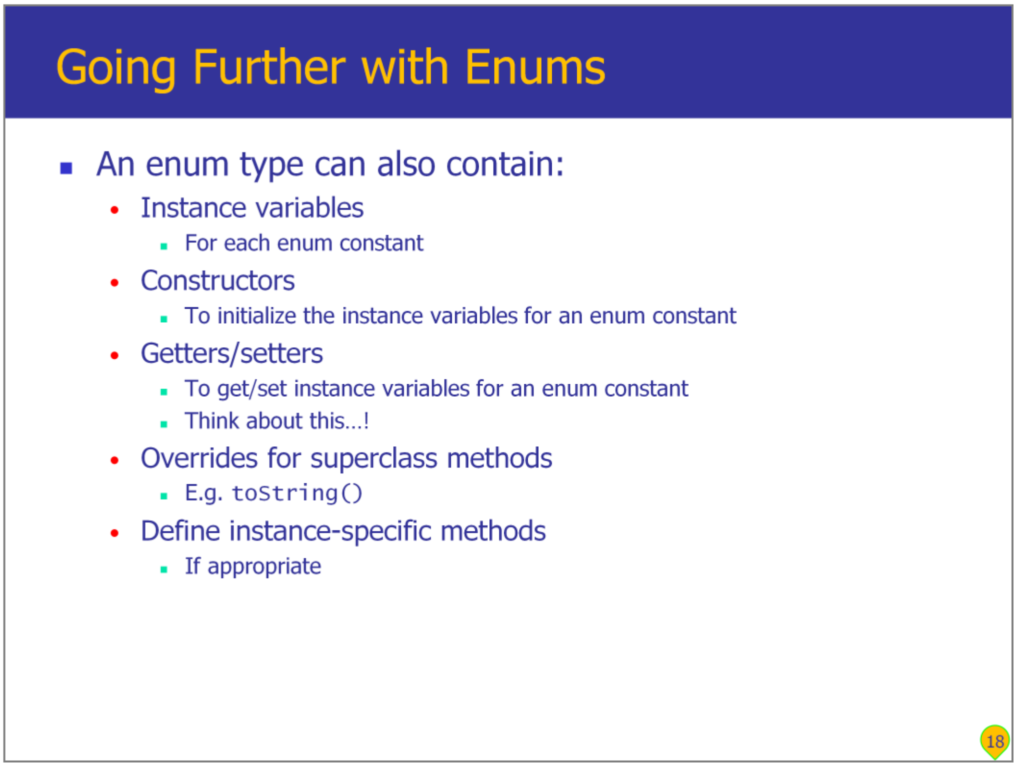
```
Direction southPole = Direction.SOUTH;
System.out.println("southPole string:  " + southPole.toString());
System.out.println("southPole ordinal: " + southPole.ordinal());
```

```
southPole string:  SOUTH
southPole ordinal: 1
```

17

Using enums in client code is easy. Simply use the enum type, followed by a dot, followed by the enum field you're interested in. For example, `Direction.SOUTH`.

Note that enum types have `equals()`, `toString()`, and `ordinal()` methods. The example in the slide shows the effect of the `toString()` and `ordinal()` methods in particular.



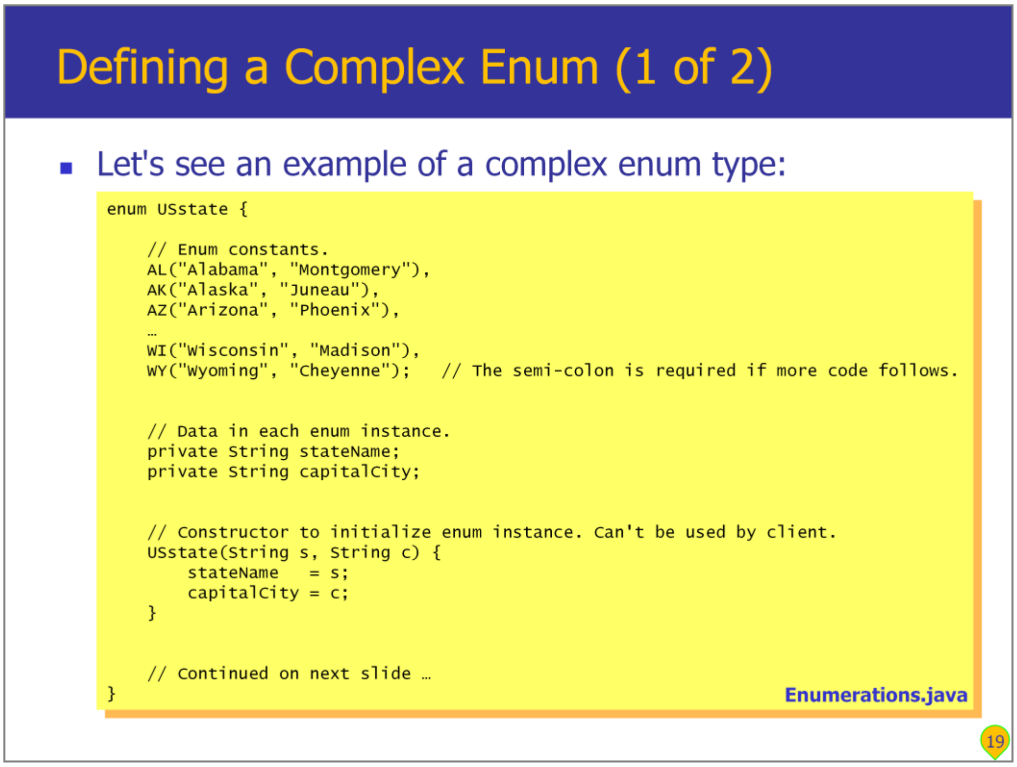
Going Further with Enums

- An enum type can also contain:
  - Instance variables
    - For each enum constant
  - Constructors
    - To initialize the instance variables for an enum constant
  - Getters/setters
    - To get/set instance variables for an enum constant
    - Think about this...!
  - Overrides for superclass methods
    - E.g. `toString()`
  - Define instance-specific methods
    - If appropriate

18

The examples on the previous slides have shown simple usage of enums. In most cases, that's all you need to know.

However, if you dig a bit deeper, you'll find that enum types behave more like real classes. For example, an enum type can define additional instance variables to accompany each value, plus constructors and methods to access these additional instance variables. This is a niche area, but it's also very powerful, so the following slide shows how it works.



Defining a Complex Enum (1 of 2)

- Let's see an example of a complex enum type:

```
enum USstate {

    // Enum constants.
    AL("Alabama", "Montgomery"),
    AK("Alaska", "Juneau"),
    AZ("Arizona", "Phoenix"),
    …
    WI("Wisconsin", "Madison"),
    WY("Wyoming", "Cheyenne");   // The semi-colon is required if more code follows.


    // Data in each enum instance.
    private String stateName;
    private String capitalCity;


    // Constructor to initialize enum instance. Can't be used by client.
    USstate(String s, String c) {
        stateName   = s;
        capitalCity = c;
    }


    // Continued on next slide …
}
```

Enumerations.java

19

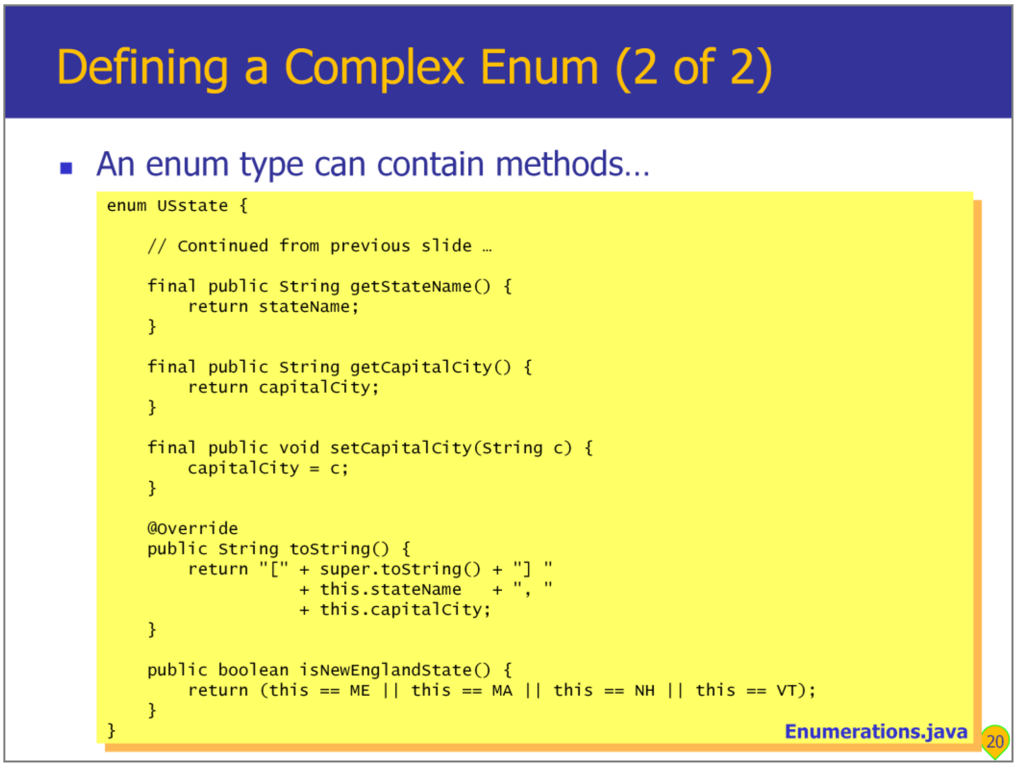
This enum is much more interesting than the one we looked at earlier. This enum represents all the states in the USA. Each state is denoted by a name such as `AL`, `AK`, `AZ`, etc.

Under the covers, each of these names represents a separate object of the `USstate` type. Each object also contains two additional instance variables:

- `stateName` indicates the name of the state, e.g. "Alabama".
- `capitalCity` indicates the capital city in that state, e.g. "Montgomery"

The `USstate` class defines a constructor that initializes these instance variables for a particular state. This explains the syntax for the enum values - each of these is effectively creating a new object and passing parameters into the constructor:

```
AL("Alabama", "Montgomery"),
AK("Alaska", "Juneau"),
AZ("Arizona", "Phoenix"),
…
```



Defining a Complex Enum (2 of 2)

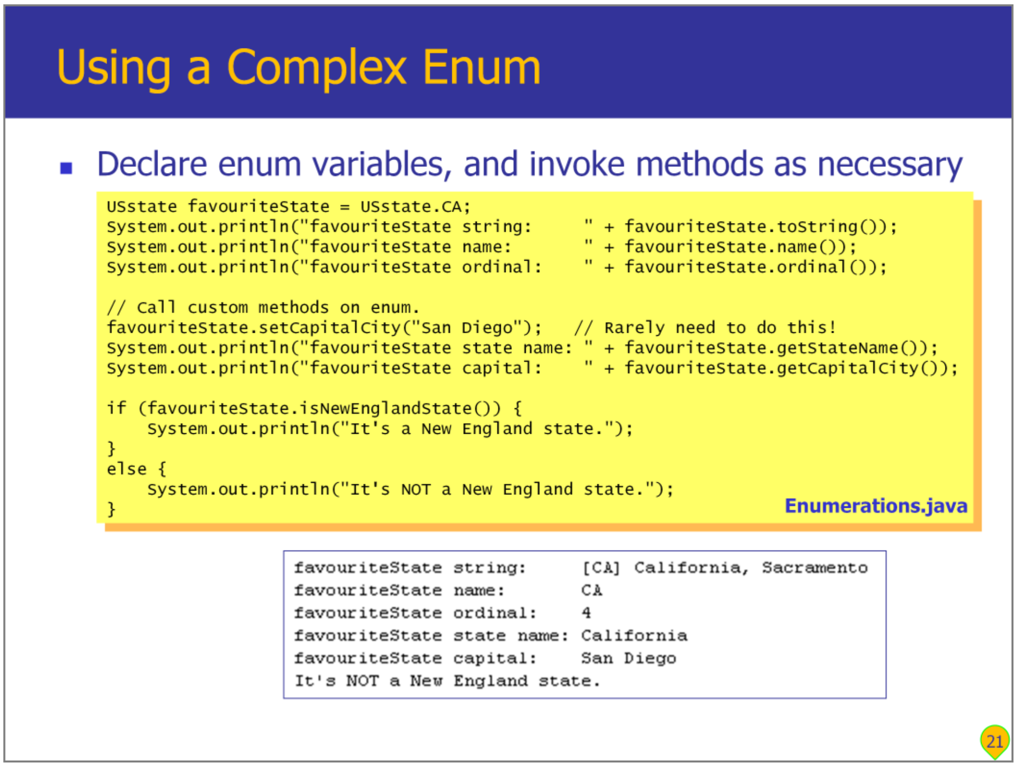
- An enum type can contain methods…

```
enum USstate {

    // Continued from previous slide …

    final public String getStateName() {
        return stateName;
    }

    final public String getCapitalCity() {
        return capitalCity;
    }

    final public void setCapitalCity(String c) {
        capitalCity = c;
    }

    @Override
    public String toString() {
        return "[" + super.toString() + "] "
                   + this.stateName   + ", "
                   + this.capitalCity;
    }

    public boolean isNewEnglandState() {
        return (this == ME || this == MA || this == NH || this == VT);
    }
}
```

Enumerations.java

20

This slide continues the `USstate` enum definition. Here, we show how to define additional instance variables in the enum type. The following slide shows how to use these capabilities in client code.



Using a Complex Enum

- Declare enum variables, and invoke methods as necessary

```
USstate favouriteState = USstate.CA;
System.out.println("favouriteState string:     " + favouriteState.toString());
System.out.println("favouriteState name:       " + favouriteState.name());
System.out.println("favouriteState ordinal:    " + favouriteState.ordinal());

// Call custom methods on enum.
favouriteState.setCapitalCity("San Diego");   // Rarely need to do this!
System.out.println("favouriteState state name: " + favouriteState.getStateName());
System.out.println("favouriteState capital:    " + favouriteState.getCapitalCity());

if (favouriteState.isNewEnglandState()) {
    System.out.println("It's a New England state.");
}
else {
    System.out.println("It's NOT a New England state.");
}
```

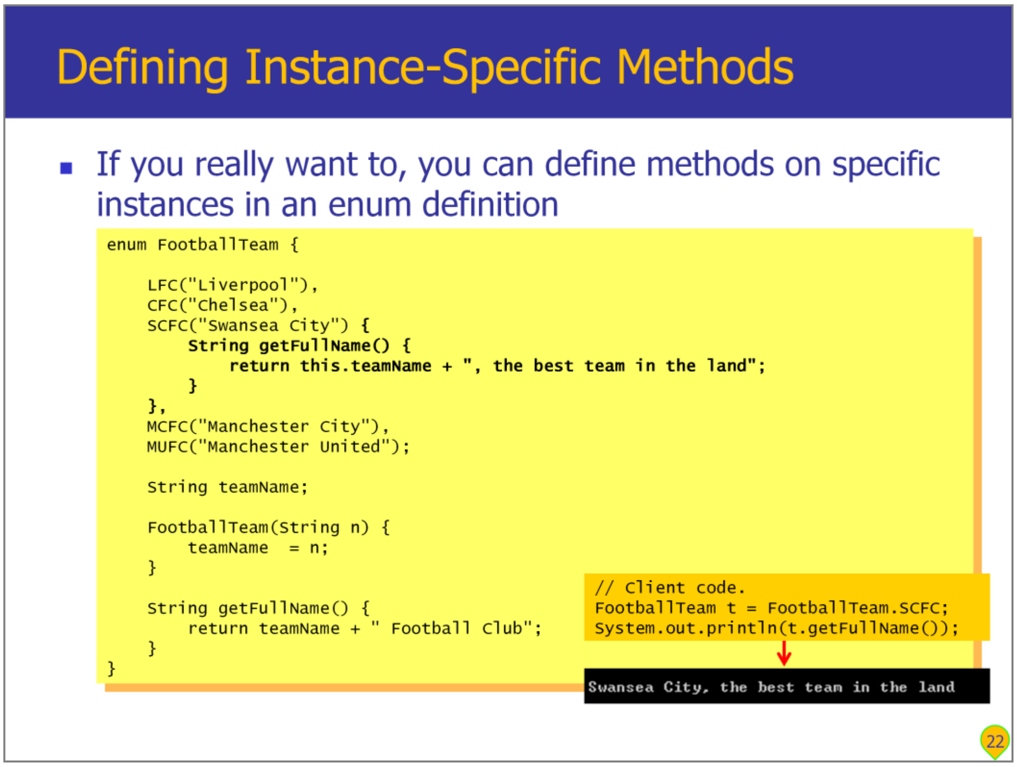
Enumerations.java

```
favouriteState string:     [CA] California, Sacramento
favouriteState name:       CA
favouriteState ordinal:    4
favouriteState state name: California
favouriteState capital:    San Diego
It's NOT a New England state.
```

21

This slide shows how to use the `USstate` enum type. Note the following points:

- First, we declare a `USstate` variable and initialize it to the `USstate.CA` object. Effectively, `USstate.CA` behaves like a key that refers to a pre-existing instance of the `USstate` enum type.
- Next, we invoke various instance methods upon the `USstate` object. At this stage, it's easy to forget that `USstate` is an enum type; it behaves more like a regular class type now.



Defining Instance-Specific Methods

- If you really want to, you can define methods on specific instances in an enum definition

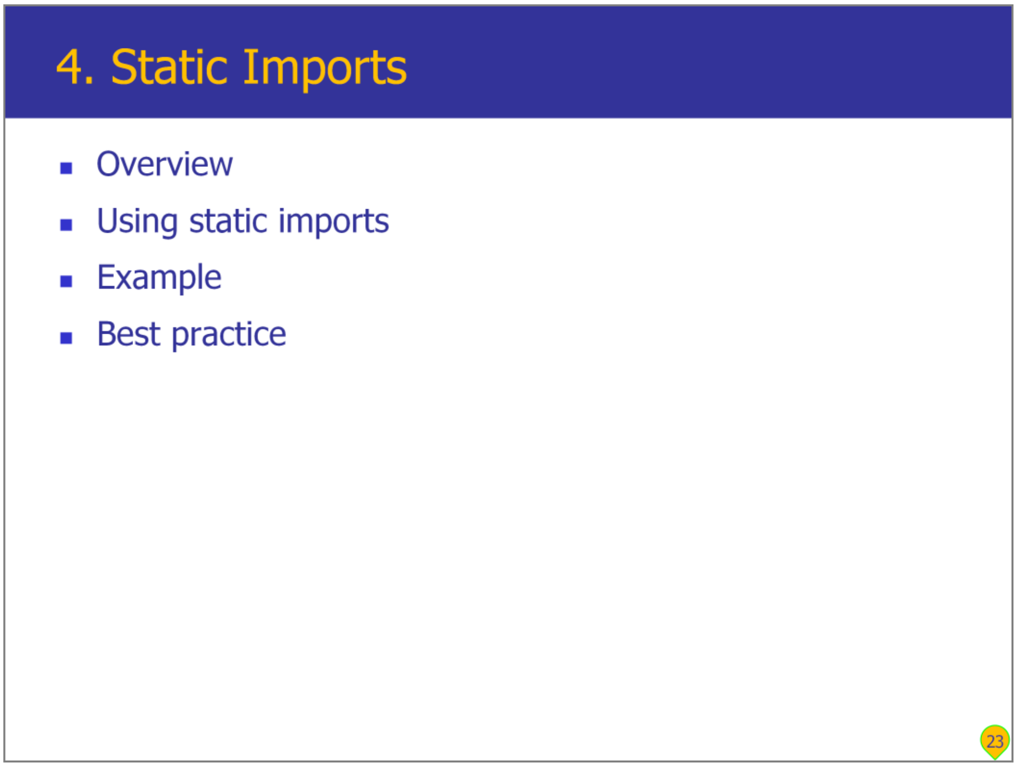
```
enum FootballTeam {

    LFC("Liverpool"),
    CFC("Chelsea"),
    SCFC("Swansea City") {
        String getFullName() {
            return this.teamName + ", the best team in the land";
        }
    },
    MCFC("Manchester City"),
    MUFC("Manchester United");

    String teamName;

    FootballTeam(String n) {
        teamName  = n;
    }

    String getFullName() {
        return teamName + " Football Club";
    }
}
```

```
// Client code.
FootballTeam t = FootballTeam.SCFC;
System.out.println(t.getFullName());
```

```
Swansea City, the best team in the land
```

22

As an advanced technique, it's also possible to define additional methods on a specific instance in an enum definition. This is useful if you need specialized behaviour for a particular instance; it’s a bit like overriding within the enum.

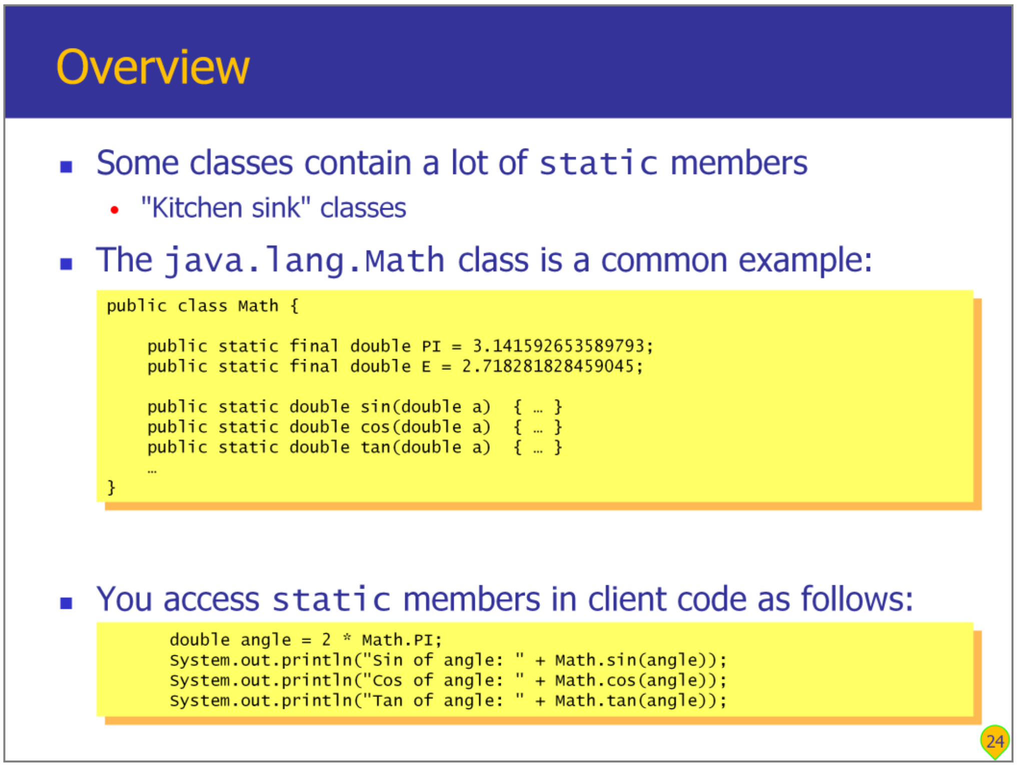


4. Static Imports

- Overview
- Using static imports
- Example
- Best practice

23

We conclude this chapter with static imports. This is probably the simplest mechanism we've covered in this chapter, so you can begin to relax now ☺.



Overview

- Some classes contain a lot of `static` members
  - "Kitchen sink" classes
- The `java.lang.Math` class is a common example:

```
public class Math {

    public static final double PI = 3.141592653589793;
    public static final double E = 2.718281828459045;

    public static double sin(double a)  { … }
    public static double cos(double a)  { … }
    public static double tan(double a)  { … }
    …
}
```

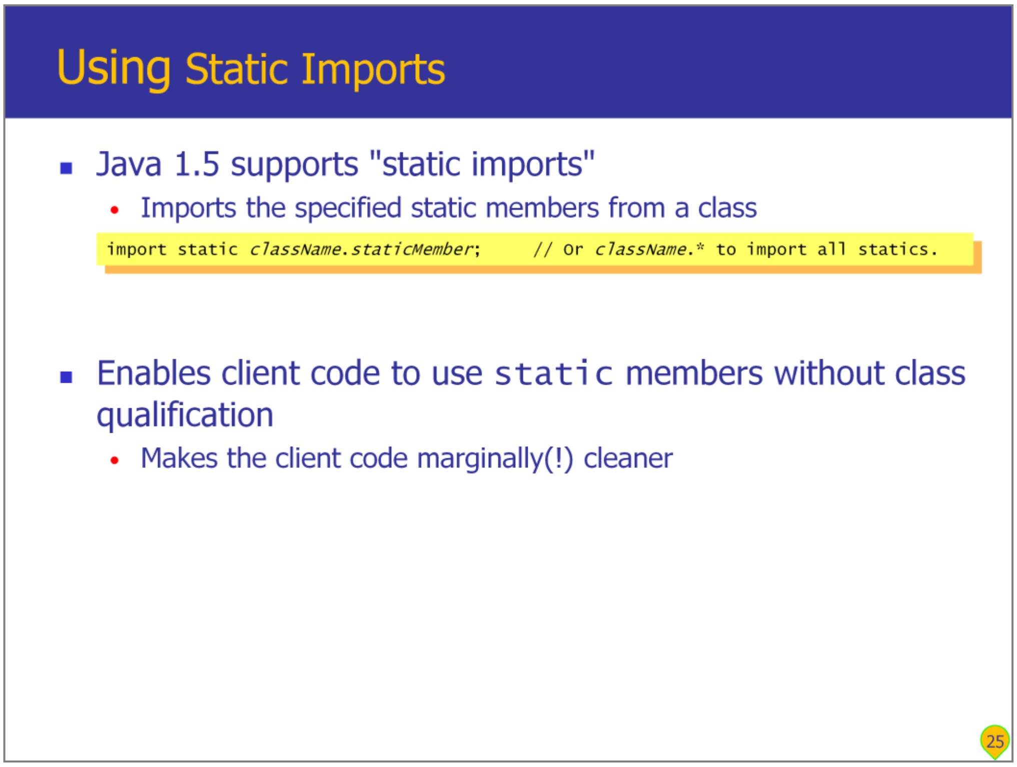
- You access `static` members in client code as follows:

```
      double angle = 2 * Math.PI;
      System.out.println("Sin of angle: " + Math.sin(angle));
      System.out.println("Cos of angle: " + Math.cos(angle));
      System.out.println("Tan of angle: " + Math.tan(angle));
```

24

Some classes in Java contain a lot of `static` methods and/or `static` data members. The `Math` class is a good example - as illustrated in the upper code box in the slide.

To use `static` members in client code, you can use the class name, followed by a dot, followed by the name of the `static` member. The lower code box in the slide shows how to do this.



Using Static Imports

- Java 1.5 supports "static imports"
  - Imports the specified static members from a class

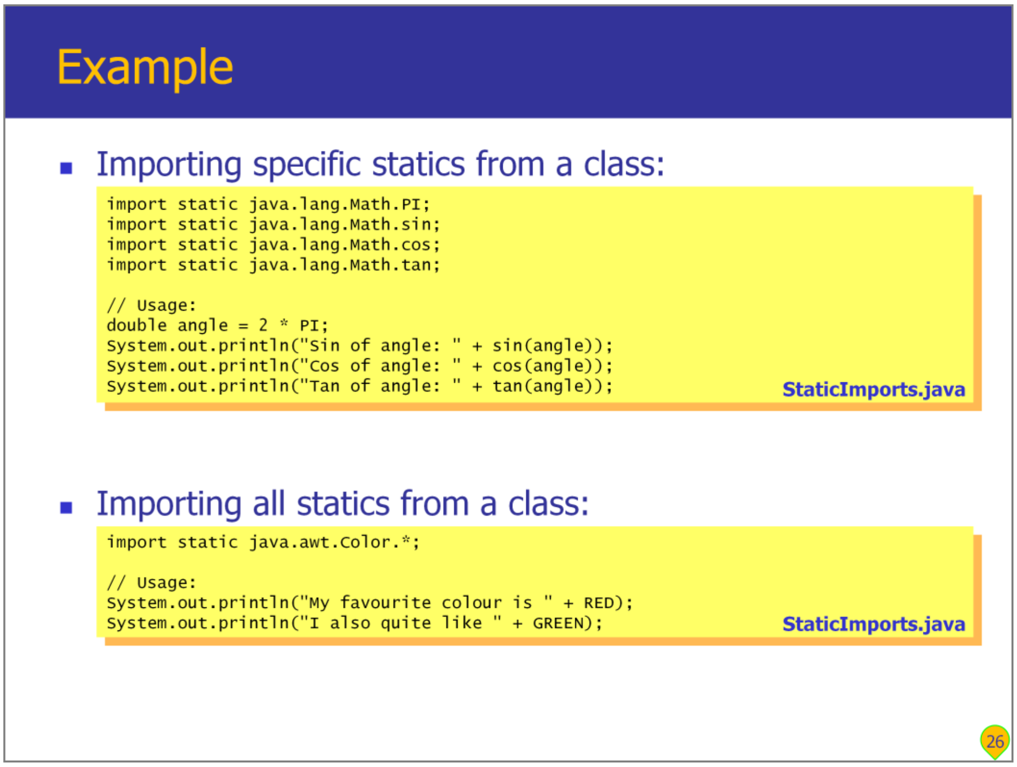
```
import static className.staticMember;     // Or className.* to import all statics.
```

- Enables client code to use `static` members without class qualification
  - Makes the client code marginally(!) cleaner

25

Java 1.5 onwards supports the concept of static imports. At the top of your code file (just after your `package` statement), you can define `import` statements that import some or all `static` members from another class.

This brings the `static` members into scope in your code file. You can then use these `static` members directly in your code, without having to prefix them with the name of the class in which they are defined.



Example

- Importing specific statics from a class:

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;
import static java.lang.Math.cos;
import static java.lang.Math.tan;

// Usage:
double angle = 2 * PI;
System.out.println("Sin of angle: " + sin(angle));
System.out.println("Cos of angle: " + cos(angle));
System.out.println("Tan of angle: " + tan(angle));
```

StaticImports.java

- Importing all statics from a class:

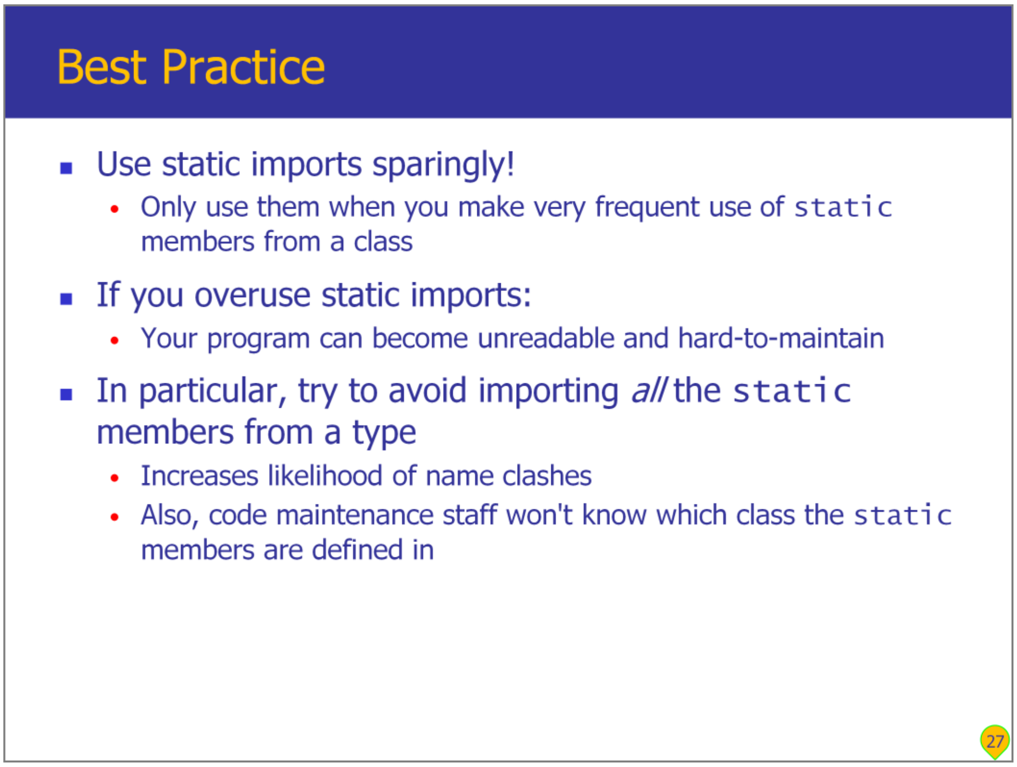
```
import static java.awt.Color.*;

// Usage:
System.out.println("My favourite colour is " + RED);
System.out.println("I also quite like " + GREEN);
```

StaticImports.java

26

This slide shows how to use static imports. Note the following points:

- The upper code box shows how to import `static` methods one at a time, and then use these methods directly in the client code (i.e. without a `Math.` prefix).
- The lower code box shows how to import all the `static` members from a class in one go, and then then use these members directly in the client code (i.e. without a `Color.` prefix).



Best Practice

- Use static imports sparingly!
  - Only use them when you make very frequent use of `static` members from a class
- If you overuse static imports:
  - Your program can become unreadable and hard-to-maintain
- In particular, try to avoid importing *all* the `static` members from a type
  - Increases likelihood of name clashes
  - Also, code maintenance staff won't know which class the `static` members are defined in

27

Static imports are very handy, if used shrewdly. For example, if you're writing JUnit unit tests, static imports considerably simplify how you use `static` methods such as `assertEquals()`.

Just be careful you don't overdo it. If you import `static` members from several classes, you run the risk of name collisions between `static` members imported from different classes.

Any Questions?