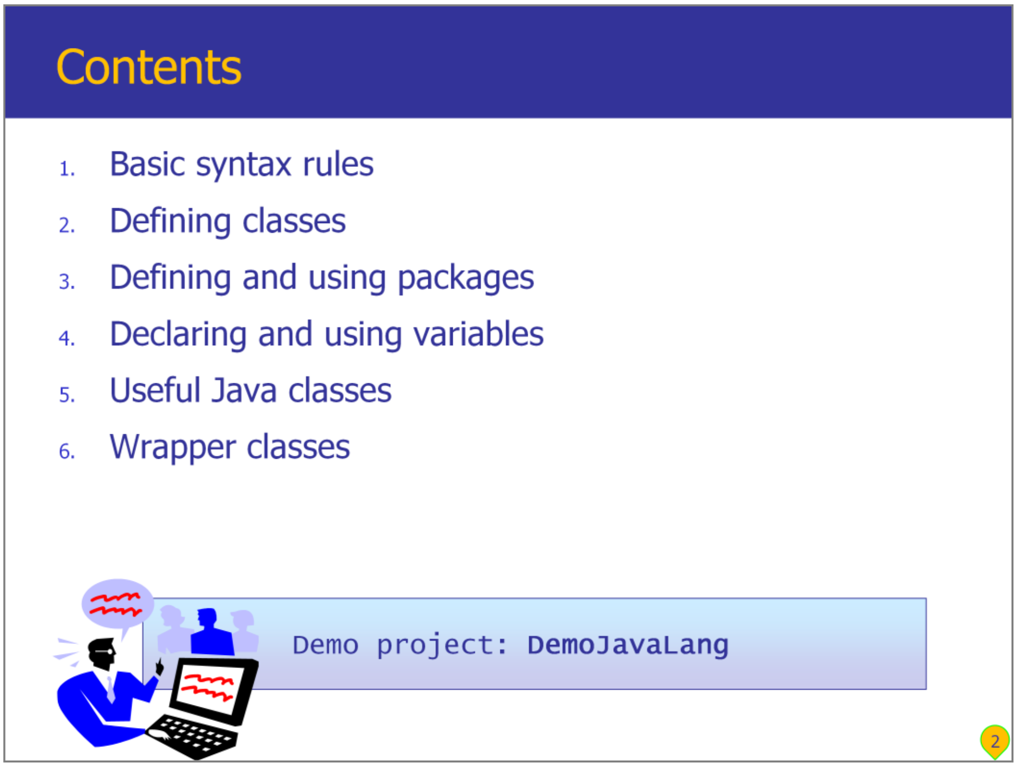
Java Language Fundamentals

This chapter takes a look at fundamental Java syntax rules. We'll also provide a quick introduction to some of the common classes in the Java library, because you can't really do anything in Java without using some of these classes.



Contents

Demo project: `DemoJavaLang`

2

Section 1 in this chapter describes the basic syntax rules such as case sensitivity, comments, the use of semi-colons to terminate statements, etc.

Section 2 provides a very high-level introduction to defining classes. We'll revisit this topic in much more detail later in the course.

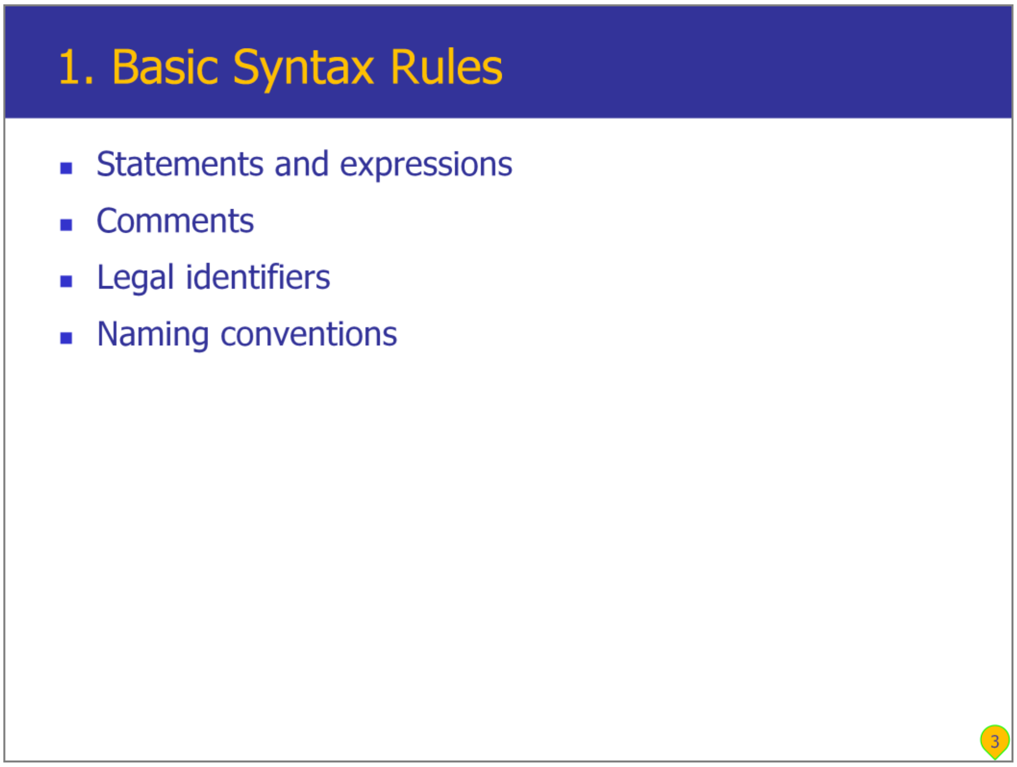
Section 3 introduces the concept of packages. We'll describe what they are, and how packages correspond to the folder hierarchy in your project.

Section 4 summarizes the rules for declaring variables in Java. As part of this discussion, we'll describe how Java differentiates between primitive types (such as `int` and `float`) and class types.

Section 5 gives you an overview of some common types in Java, such as `String` and `Math`.

Section 6 introduces the concept of wrapper classes. This term has a specific meaning in Java – basically every primitive type (such as `int`) has a corresponding wrapper class (such as `Integer`). The wrapper class contains useful methods for handling the corresponding primitive type.

The demos for chapter are located in the `DemoJavaLang` project. If you haven't already done so, you can import this project into your Eclipse workspace.

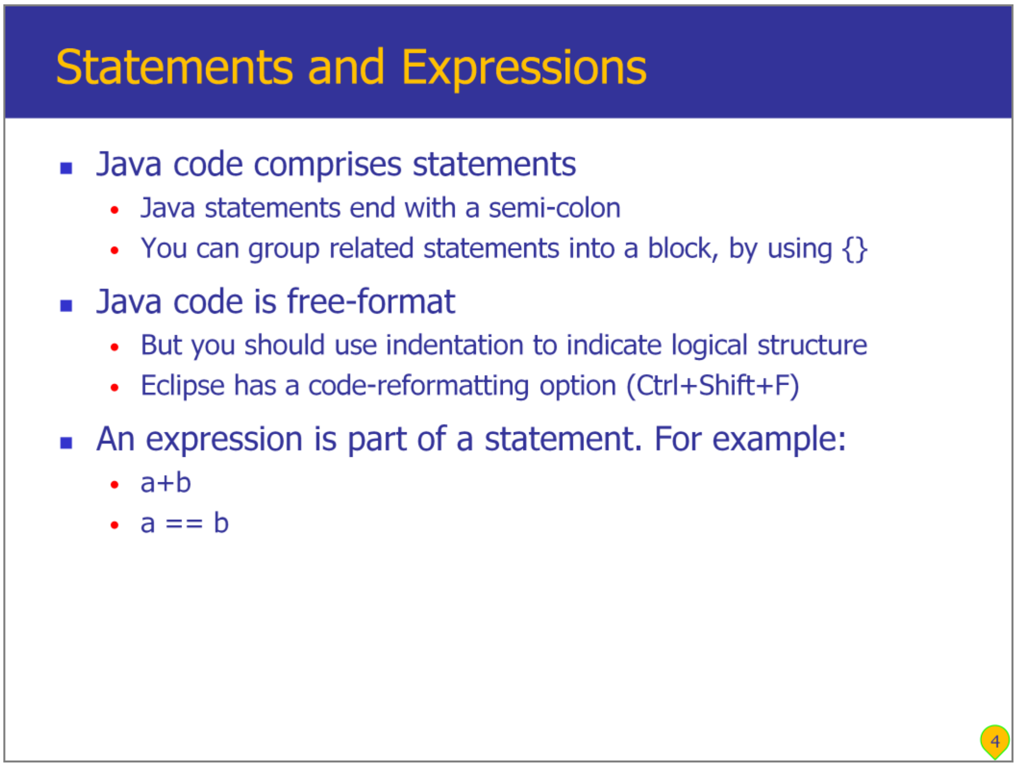


1. Basic Syntax Rules

- Statements and expressions
- Comments
- Legal identifiers
- Naming conventions

3

This section describes the fundamental rules of Syntax in Java code. The Java compiler is very fussy, so it's important you obey these rules… otherwise you'll get errors when you try to compile your code.



Statements and Expressions

- Java code comprises statements
  - Java statements end with a semi-colon
  - You can group related statements into a block, by using {}
- Java code is free-format
  - But you should use indentation to indicate logical structure
  - Eclipse has a code-reformatting option (Ctrl+Shift+F)
- An expression is part of a statement. For example:
  - a+b
  - a == b

4

In Java, statements must be terminated with a semi-colon. It's ok for statements to span over multiple lines, but there must be a semi-colon at the end.

For example, here's a simple statement that outputs a person's name. The example assumes we've defined variables called `fname` and `lname` to hold the person's first and last names:

```
    System.out.println("My name is " + fname + " " + lname);
```

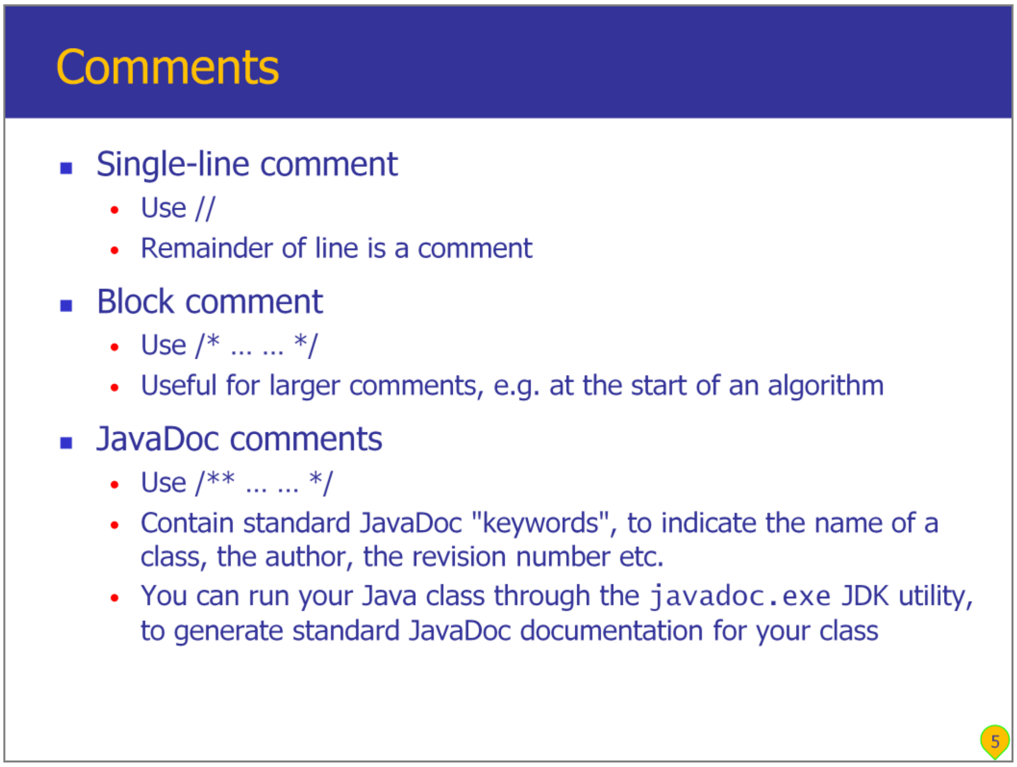
If you prefer, you can split this statement over several lines – just remember to put the semi-colon at the end:

```
   System.out.println("My name is " +
                      fname +
                      " " +
                      lname);
```

Eclipse (and other IDEs) have tools that help you to auto-format your code. For example, in Eclipse you can use the Ctrl+Shift+F shortcut to reformat your code. There are lots of other options available as well – right-click in the code window in Eclipse, and select Source from the popup menu.

A couple of other points worth mentioning…

- You can create a compound statement by using `{}`, e.g. to define loop bodies. In this case, you don't put a semi-colon after the closing `}`.
- A statement can contain any number of expressions, e.g. `a+b`, `a==b`, etc.



Comments

- Single-line comment
  - Use //
  - Remainder of line is a comment
- Block comment
  - Use /* … … */
  - Useful for larger comments, e.g. at the start of an algorithm
- JavaDoc comments
  - Use /** … … */
  - Contain standard JavaDoc "keywords", to indicate the name of a class, the author, the revision number etc.
  - You can run your Java class through the `javadoc.exe` JDK utility, to generate standard JavaDoc documentation for your class

5

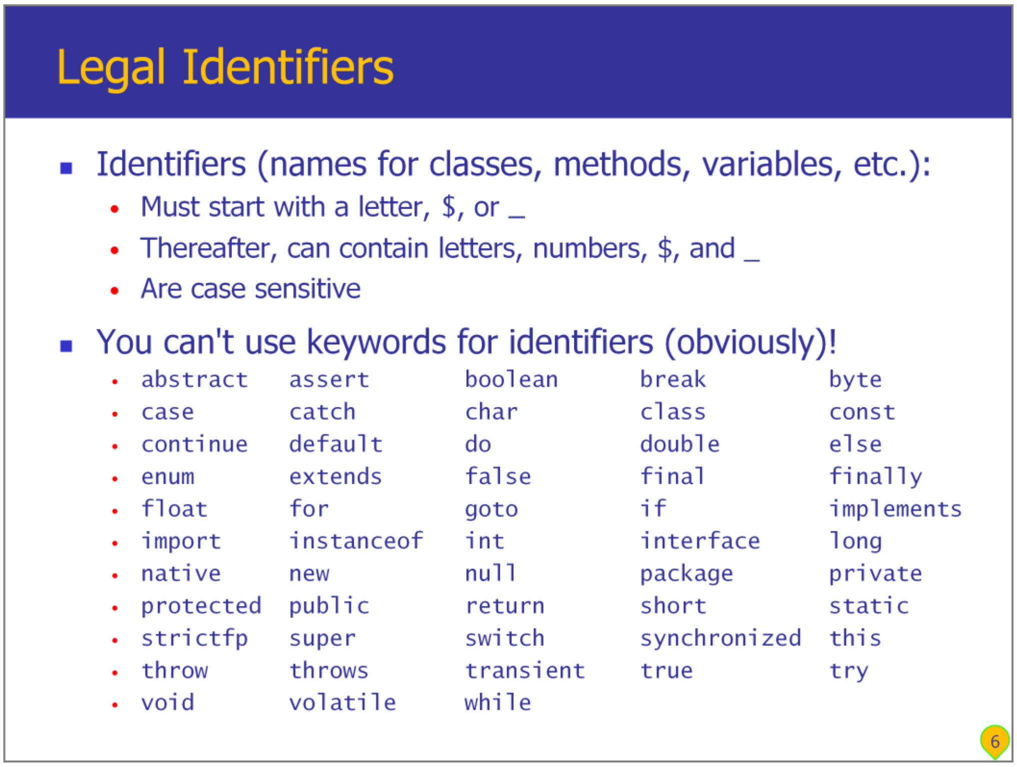
Comments are important, as long as you don't overdo it! There are three different ways to define comments in Java:

- `//`
  This is a single-line comment. Everything afterwards, to the end of the line, is treated as a comment.

- `/* … */`
  This is a block comment. Everything between the start and end is a comment. Note that block comments do not nest – if you define a block comment inside another block comment, the first */ will terminate the entire comment.

- `/** … */`
  This is a JavaDoc comment. This is a Java-specific mechanism that allows you to embed metadata in your code, e.g. author, method parameter info, etc. For more information about JavaDoc comments, see the following web site:

  http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html



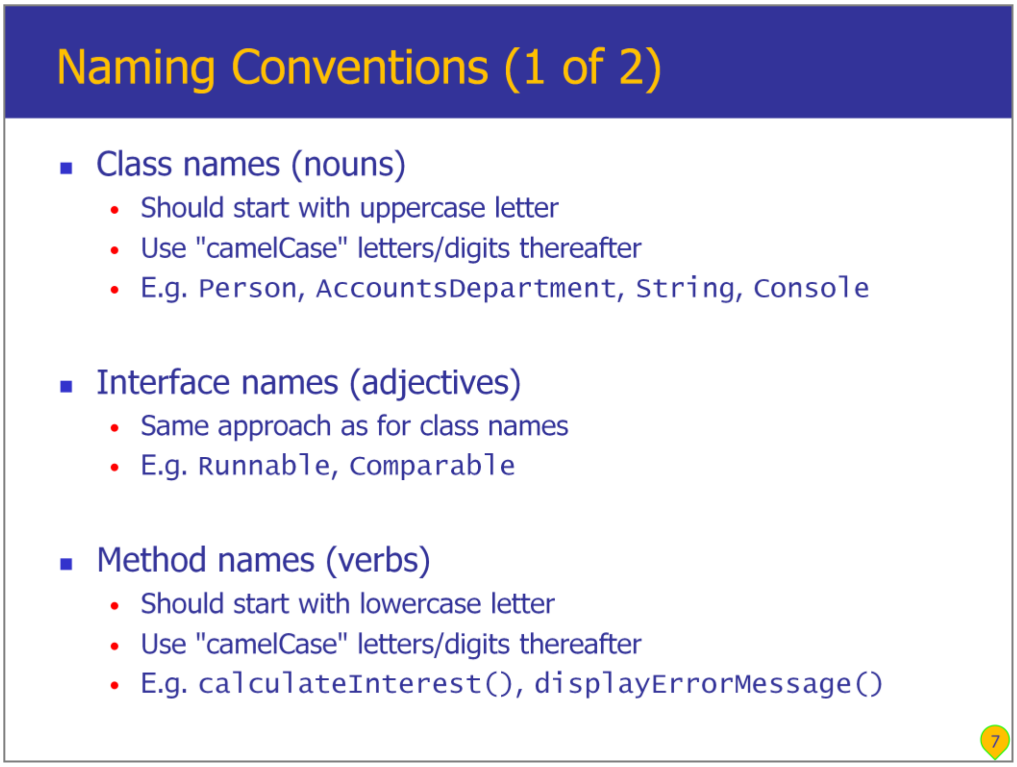
Legal Identifiers

- Identifiers (names for classes, methods, variables, etc.):
  - Must start with a letter, $, or _
  - Thereafter, can contain letters, numbers, $, and _
  - Are case sensitive
- You can't use keywords for identifiers (obviously)!

`abstract`	`assert`	`boolean`	`break`	`byte`
`case`	`catch`	`char`	`class`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extends`	`false`	`final`	`finally`
`float`	`for`	`goto`	`if`	`implements`
`import`	`instanceof`	`int`	`interface`	`long`
`native`	`new`	`null`	`package`	`private`
`protected`	`public`	`return`	`short`	`static`
`strictfp`	`super`	`switch`	`synchronized`	`this`
`throw`	`throws`	`transient`	`true`	`try`
`void`	`volatile`	`while`		

Java has some simple rules about how to define the names for things in your code, such as classes, methods (i.e. functions), variables, etc. The first bullet point on the slide summarizes these rules. Bear in mind that Java is case-sensitive!

Also bear in mind that certain words are predefined in Java as keywords – see the lower bullet point in the slide. Some of these are more obvious and intuitive than others, but they all have special meaning, so you can't use any of them for your own classes, variables, etc.



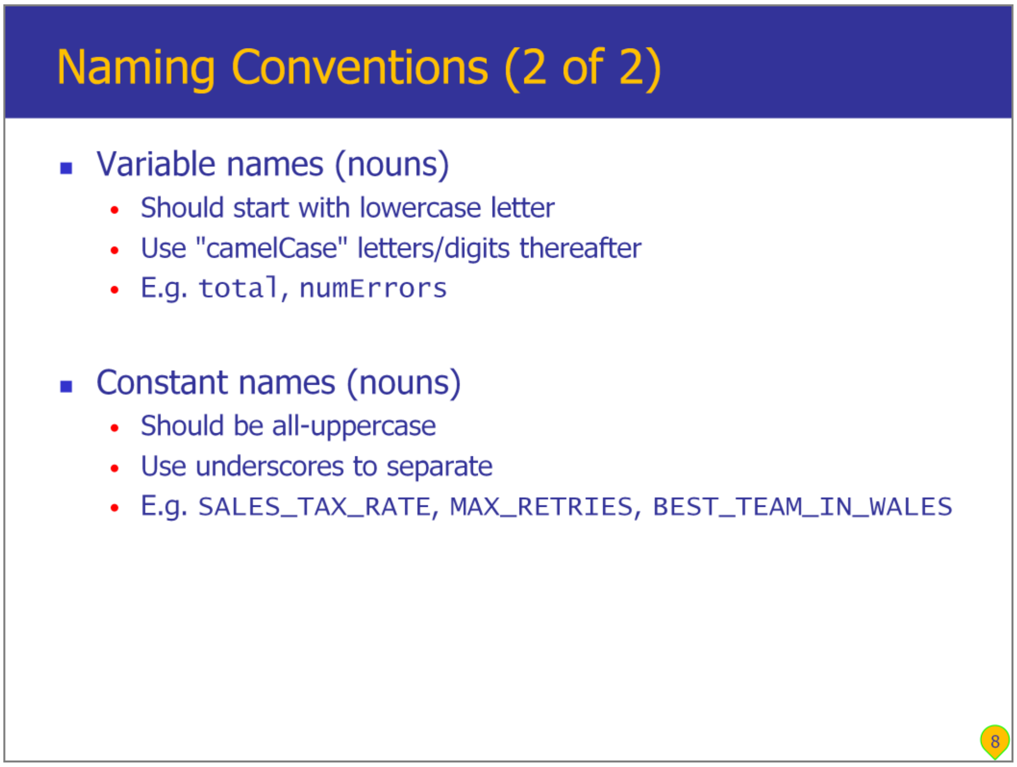
Naming Conventions (1 of 2)

- Class names (nouns)
  - Should start with uppercase letter
  - Use "camelCase" letters/digits thereafter
  - E.g. `Person`, `AccountsDepartment`, `String`, `Console`
- Interface names (adjectives)
  - Same approach as for class names
  - E.g. `Runnable`, `Comparable`
- Method names (verbs)
  - Should start with lowercase letter
  - Use "camelCase" letters/digits thereafter
  - E.g. `calculateInterest()`, `displayErrorMessage()`

7

As well as having the hard-and-fast rules specified on the previous slide, there are also several naming conventions. These naming conventions are listed in the slide above.

These conventions are optional really, but you are strongly advised to adopt them – otherwise your code will look very strange to other Java programmers.

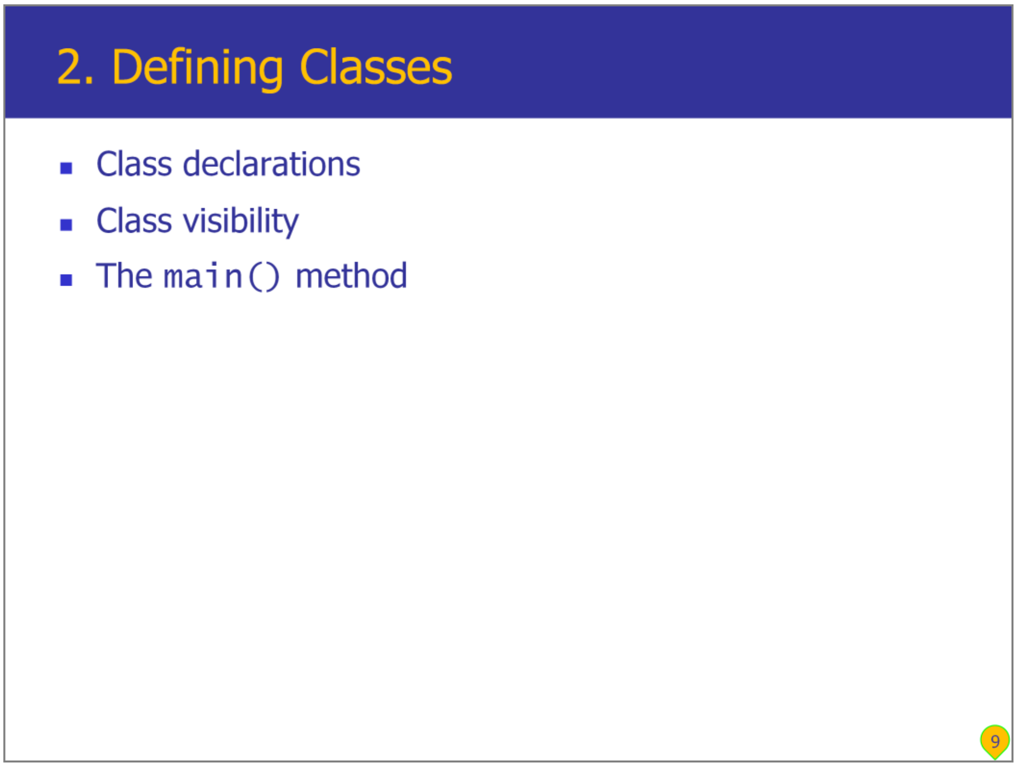


Naming Conventions (2 of 2)

- Variable names (nouns)
  - Should start with lowercase letter
  - Use "camelCase" letters/digits thereafter
  - E.g. `total`, `numErrors`
- Constant names (nouns)
  - Should be all-uppercase
  - Use underscores to separate
  - E.g. `SALES_TAX_RATE`, `MAX_RETRIES`, `BEST_TEAM_IN_WALES`

Here are some more naming conventions. We'll be using all these conventions in all the code in this course.

Note that the `BEST_TEAM_IN_WALES` is a constant, it will always be Swansea ☺.



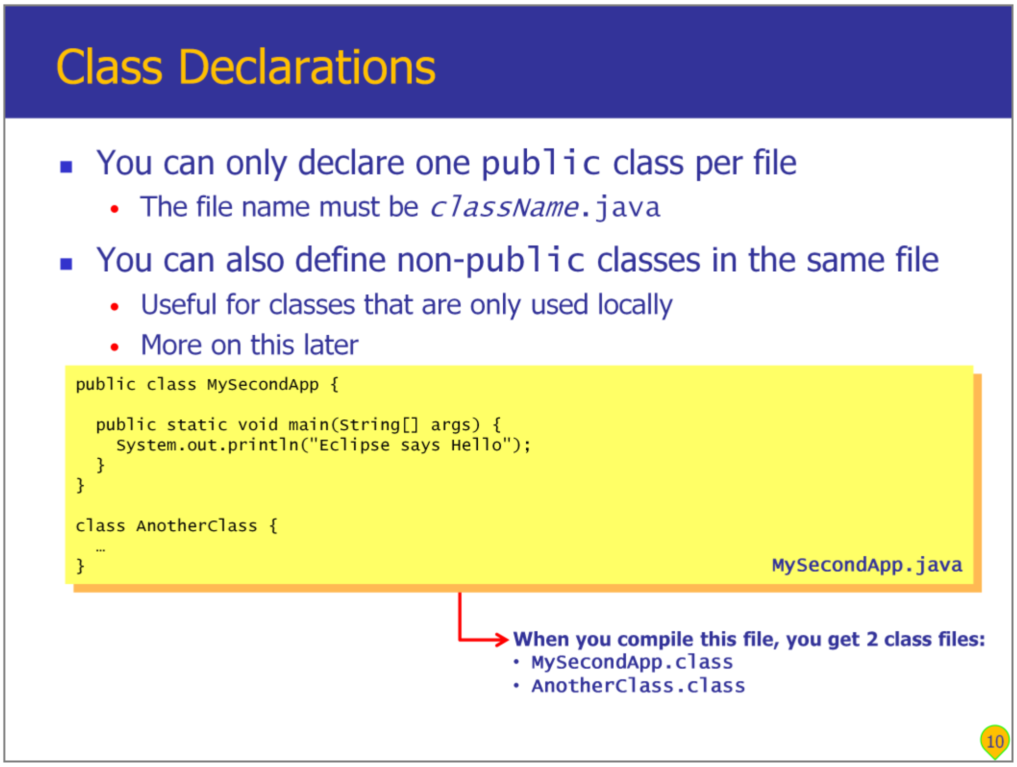
2. Defining Classes

- Class declarations
- Class visibility
- The `main()` method

9

This section runs you through a very high-level introduction about how to define classes in Java. You can't really avoid this, because everything in Java is basically a class (or something very similar, such as an interface).

We'll come back to the concept of classes in much more detail later in the course.



Class Declarations

- You can only declare one `public` class per file
  - The file name must be *`className`*`.java`
- You can also define non-`public` classes in the same file
  - Useful for classes that are only used locally
  - More on this later

```
public class MySecondApp {

  public static void main(String[] args) {
    System.out.println("Eclipse says Hello");
  }
}

class AnotherClass {
  …
}
```

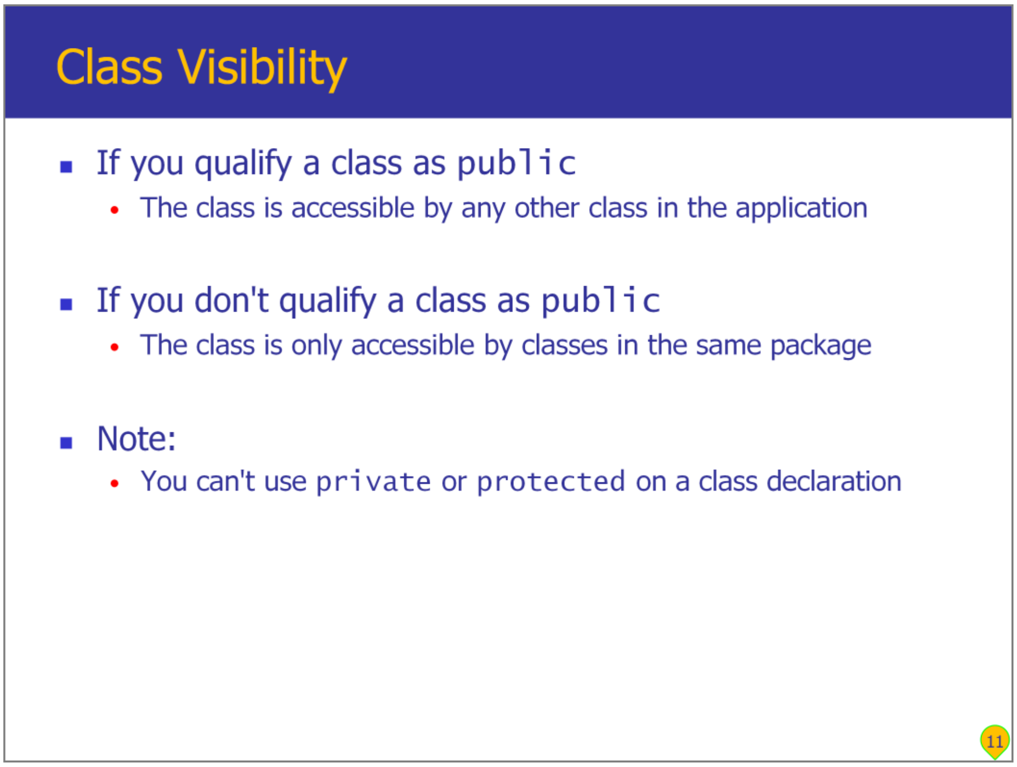
`MySecondApp.java`

When you compile this file, you get 2 class files:
- `MySecondApp.class`
- `AnotherClass.class`

10

A Java code file can only contain a single public class definition. Furthermore, the name of the class must correspond to the name of the file.

It's possible to define additional non-public classes in the same file, as illustrated in the code snippet on the slide. This example defines a public class named `MySecondApp` (which can be accessed anywhere in the application), and a non-public course named `AnotherClass`. Note that `AnotherClass` can only be accessed by other classes in the same Java file, because it isn't public.



Class Visibility

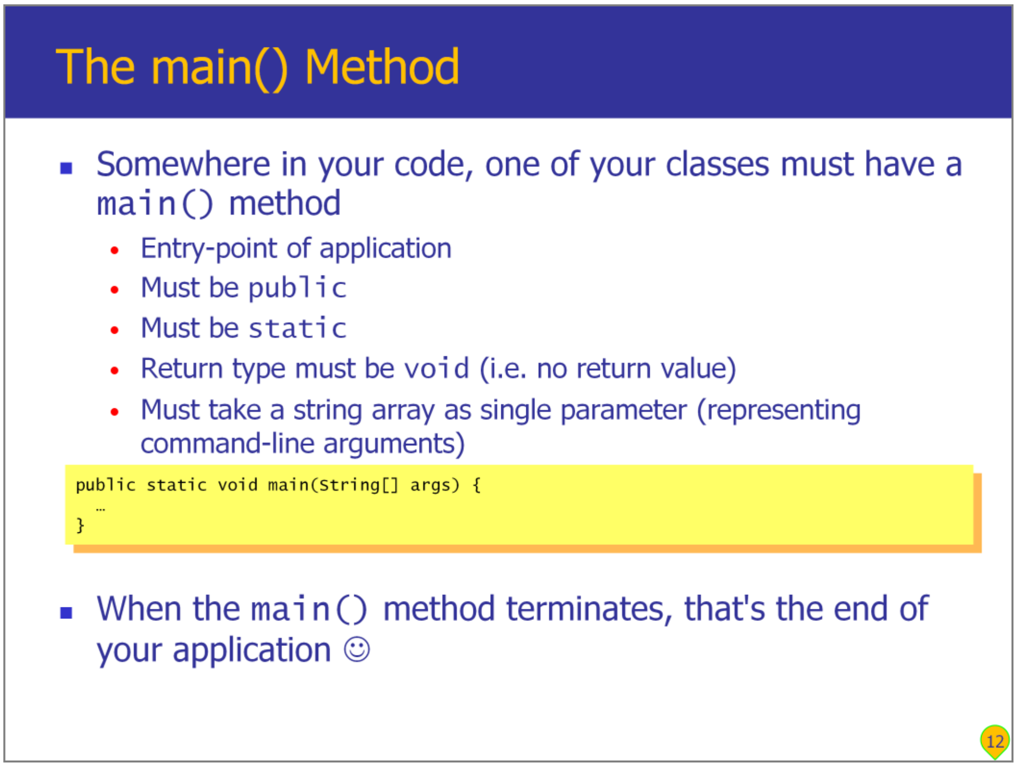
- If you qualify a class as `public`
  - The class is accessible by any other class in the application
- If you don't qualify a class as `public`
  - The class is only accessible by classes in the same package
- Note:
  - You can't use `private` or `protected` on a class declaration

11

This slide describes how you can specify the visibility of top-level classes in Java. Note that if you omit the `public` keyword on a class definition, the class has package-level visibility by default. We'll discuss packages in detail in the next section.

There are also some additional modifiers you can use with a class:

- `abstract`
  Indicates that a class is non-instantiable, which is useful for base classes in an inheritance hierarchy.
- `final`
  Indicates that a class is non-extensible, which is useful for preventing spurious inheritance.
- `strictfp`
  Indicates a class conforms to the IEEE 754 standard rules for floating points. This is an advanced keyword, which you'll probably never need to use in your own code.



The main() Method

- Somewhere in your code, one of your classes must have a `main()` method
  - Entry-point of application
  - Must be `public`
  - Must be `static`
  - Return type must be `void` (i.e. no return value)
  - Must take a string array as single parameter (representing command-line arguments)

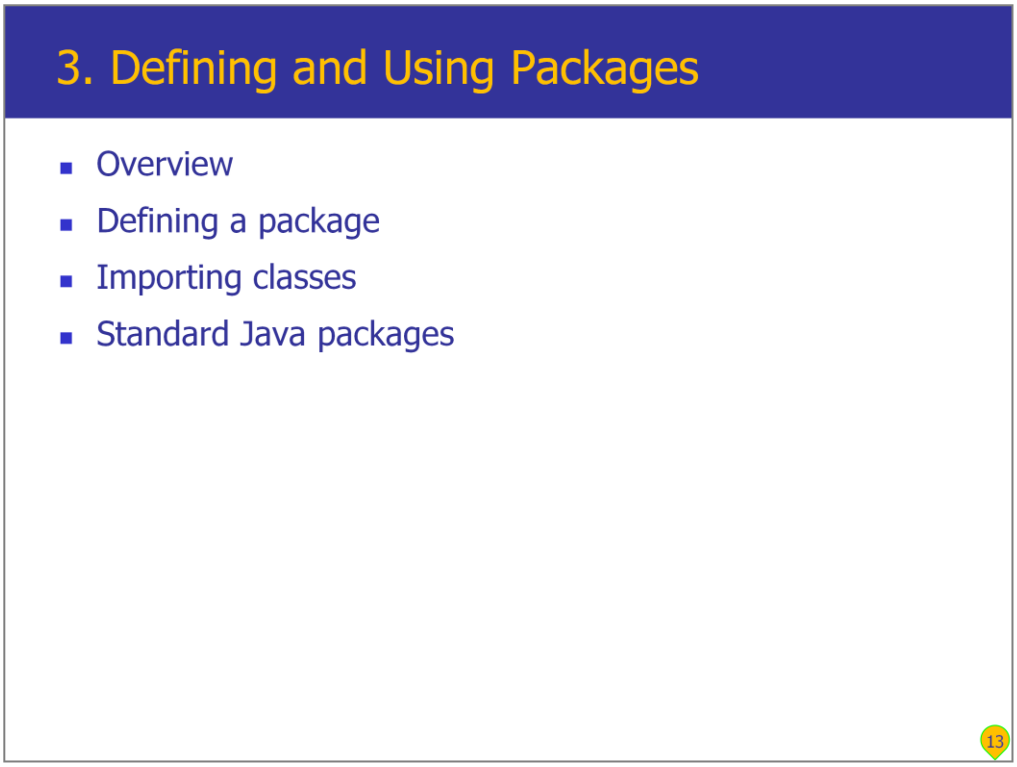
```
public static void main(String[] args) {
  …
}
```

- When the `main()` method terminates, that's the end of your application ☺

12

This slide summarizes the rules for the `main()` method, which specifies the entry point for your application. The rules for this method are very strict – you must follow all these rules.

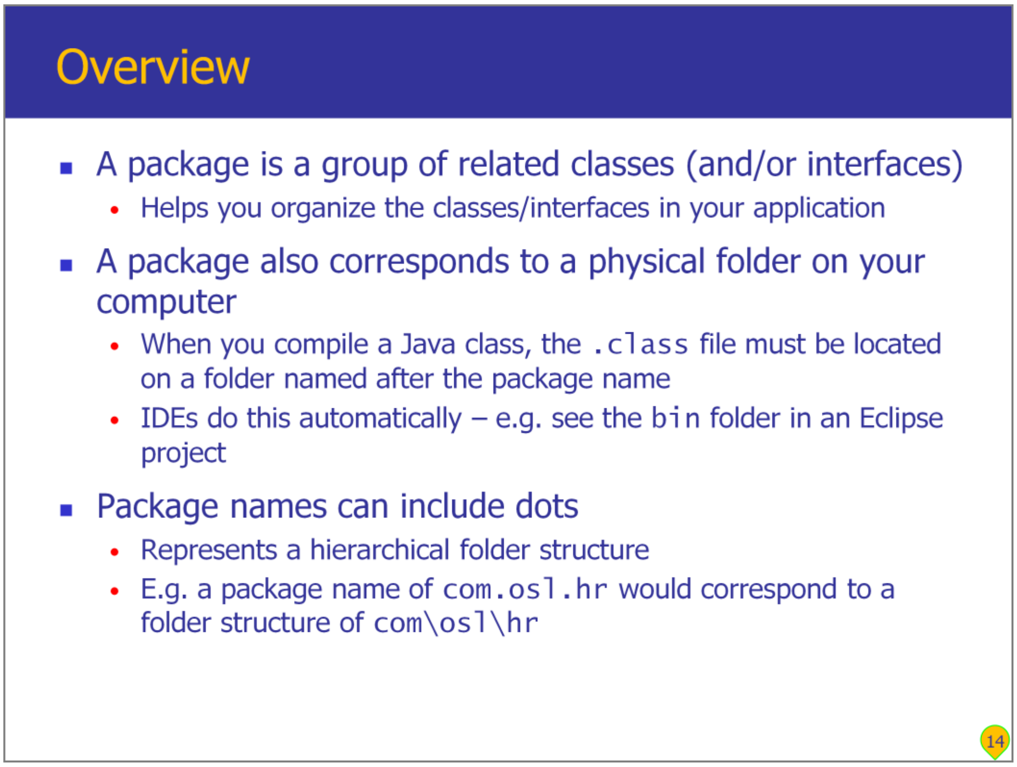
Note: Don't worry too much about the `static` keyword for now. It is important, but it's not essential you understand the details just yet. We'll discuss the `static` keyword in depth later in the course.



3. Defining and Using Packages

- Overview
- Defining a package
- Importing classes
- Standard Java packages

This section gives you the low-down on packages – what are they, why do they matter, how do you define them, and how do you import them. All will be revealed in this section.



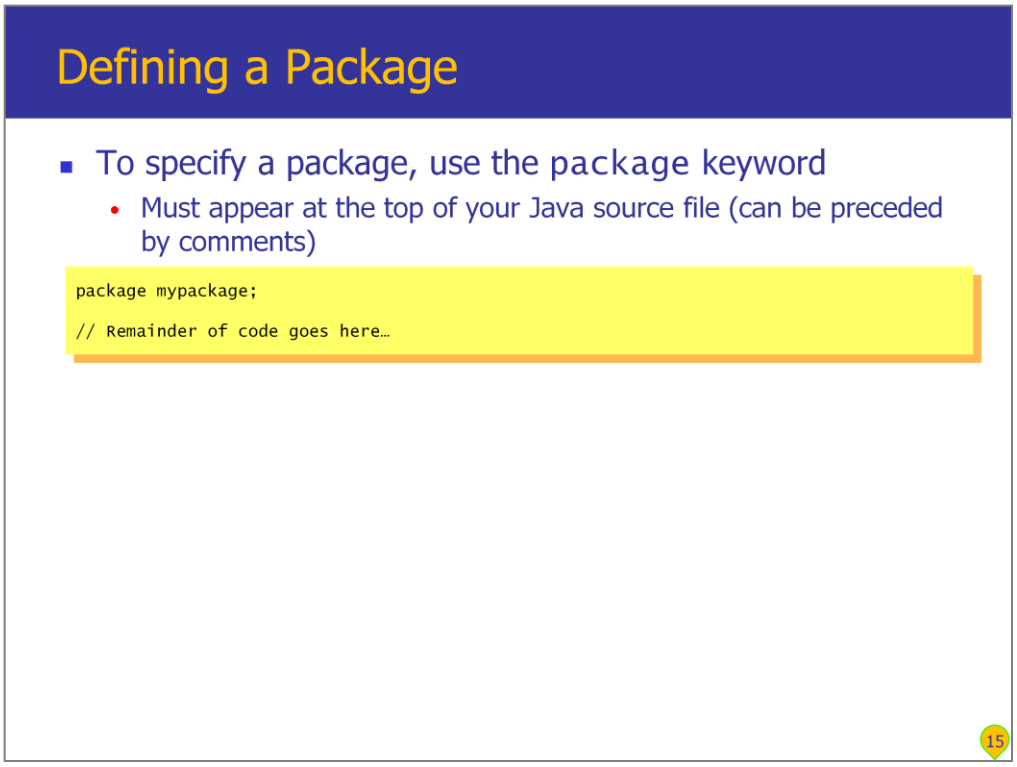
Overview

- A package is a group of related classes (and/or interfaces)
  - Helps you organize the classes/interfaces in your application
- A package also corresponds to a physical folder on your computer
  - When you compile a Java class, the `.class` file must be located on a folder named after the package name
  - IDEs do this automatically – e.g. see the `bin` folder in an Eclipse project
- Package names can include dots
  - Represents a hierarchical folder structure
  - E.g. a package name of `com.osl.hr` would correspond to a folder structure of `com\osl\hr`

14

When you create a new project in Eclipse, it doesn't have any packages initially. This means you could add your classes directly to the `src` folder… avoid this temptation! You should always put your classes into a package, to avoid any possible naming clashes that might arise in the future.

If you want to explore the physical file structure of your project in Eclipse, select the Window | Show View | Navigator menu command. The Navigator window will make it obvious that package names correspond to a hierarchical folder structure within your project.



Defining a Package

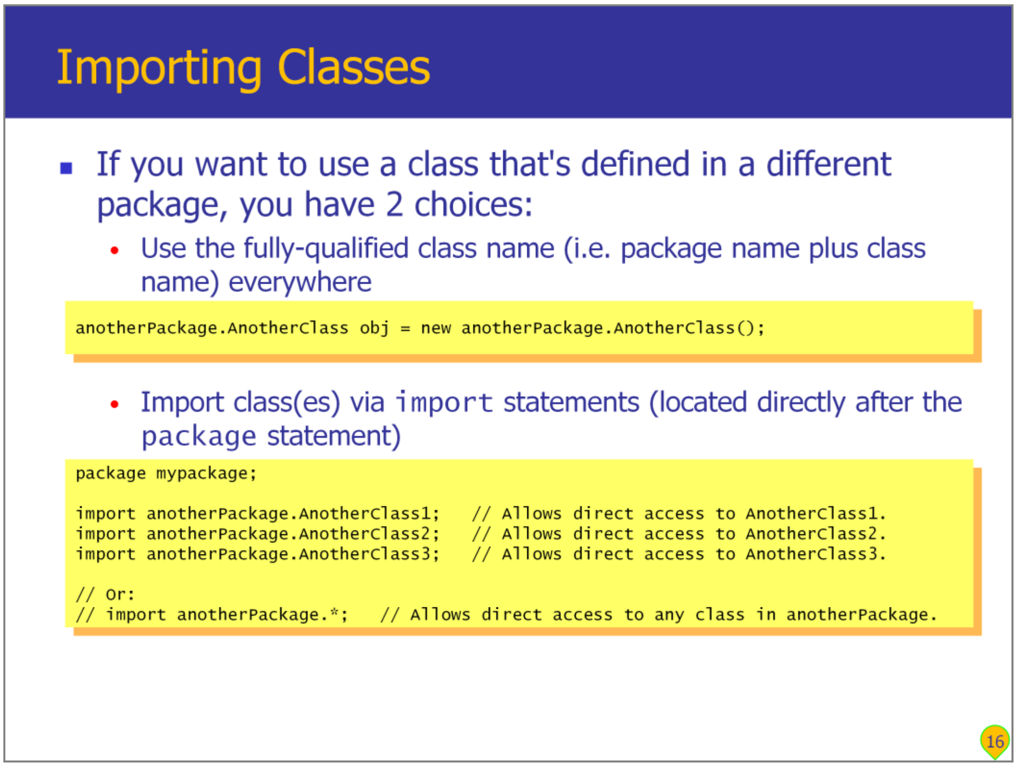
- To specify a package, use the `package` keyword
  - Must appear at the top of your Java source file (can be preceded by comments)

```
package mypackage;

// Remainder of code goes here…
```

15

To define a package for a source file, use a `package` statement as shown in the example in the slide. All classes defined in the source file will then be considered to be in that package.

Note that the `package` statement must be the first statement in the source code file.



Importing Classes

- If you want to use a class that's defined in a different package, you have 2 choices:
  - Use the fully-qualified class name (i.e. package name plus class name) everywhere

```
anotherPackage.AnotherClass obj = new anotherPackage.AnotherClass();
```

  - Import class(es) via `import` statements (located directly after the `package` statement)

```
package mypackage;

import anotherPackage.AnotherClass1;   // Allows direct access to AnotherClass1.
import anotherPackage.AnotherClass2;   // Allows direct access to AnotherClass2.
import anotherPackage.AnotherClass3;   // Allows direct access to AnotherClass3.

// Or:
// import anotherPackage.*;   // Allows direct access to any class in anotherPackage.
```

16

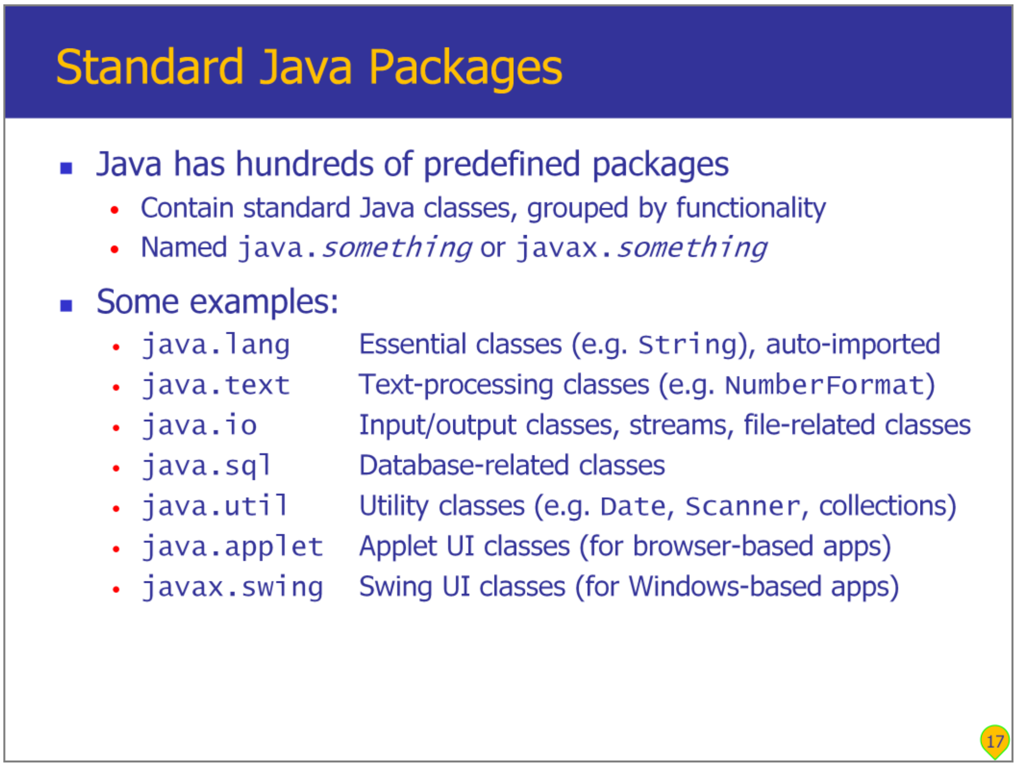
Imagine you're implementing a class that needs to use a class defined in a different package. One approach would be to use the fully qualified name of the class, i.e. `packageName.className`. The first code box in the slide shows how to do this.

An easier approach is to use an `import` statement at the top of your source code file (i.e. directly after your `package` statement). In the `import` statement, specify exactly what classes you want to import, e.g. `packageName1.className1`.

If you find yourself needing to import lots of classes from the same package, you can use a wildcard (i.e. `*`) instead of a specific class name. For example, `packageName1.*` would import all the classes from the `packageName1` package.

Here are some additional things for you to consider:

- In order to import a class, the classes must actually be on the classpath. For example, if you want to import some classes from a third-party library, your first step should be to add that library JAR file into your classpath.
- When you import a package, it just imports things from that specific package. It doesn't import anything from any sub-packages. For example, if you import `com.osl.*`, it will just import things from the `com.osl` package; it won't import sub-packages such as `com.osl.biz`, `com.osl.ui`, etc. You need separate `import` statements to import these sub-packages.



Standard Java Packages

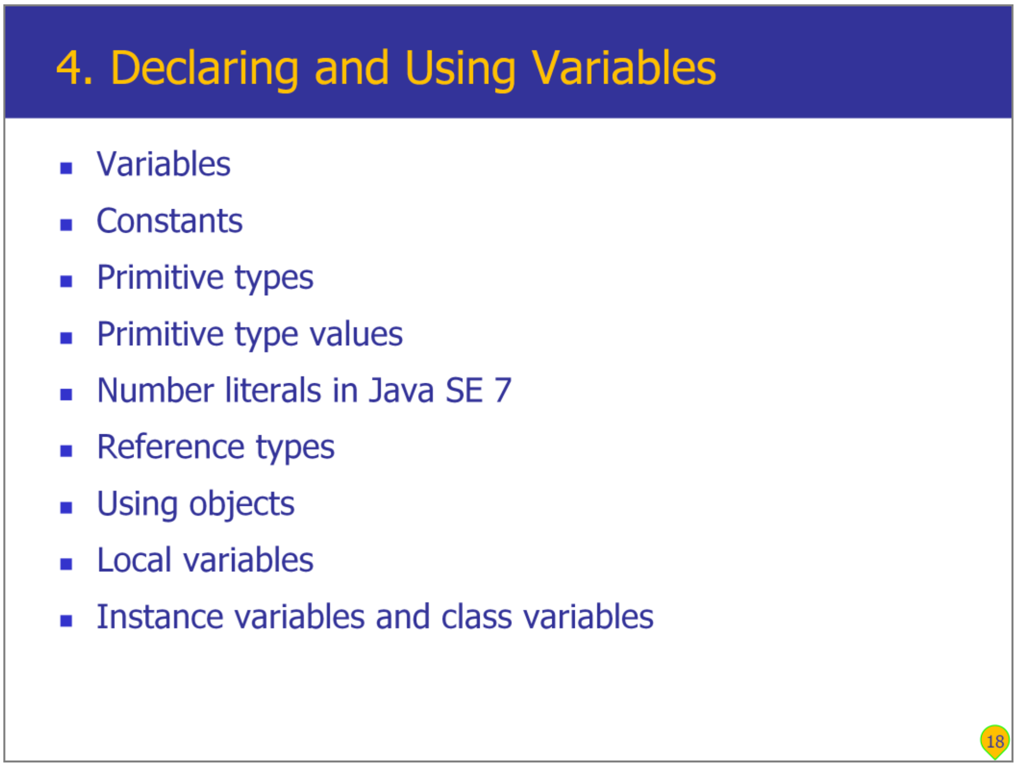
- Java has hundreds of predefined packages
  - Contain standard Java classes, grouped by functionality
  - Named `java.`*`something`* or `javax.`*`something`*
- Some examples:

`java.lang`	Essential classes (e.g. `String`), auto-imported
`java.text`	Text-processing classes (e.g. `NumberFormat`)
`java.io`	Input/output classes, streams, file-related classes
`java.sql`	Database-related classes
`java.util`	Utility classes (e.g. `Date`, `Scanner`, collections)
`java.applet`	Applet UI classes (for browser-based apps)
`javax.swing`	Swing UI classes (for Windows-based apps)

17

Java defines a large number of standard packages. All these packages start either with the `java` or `javax` package name prefix.

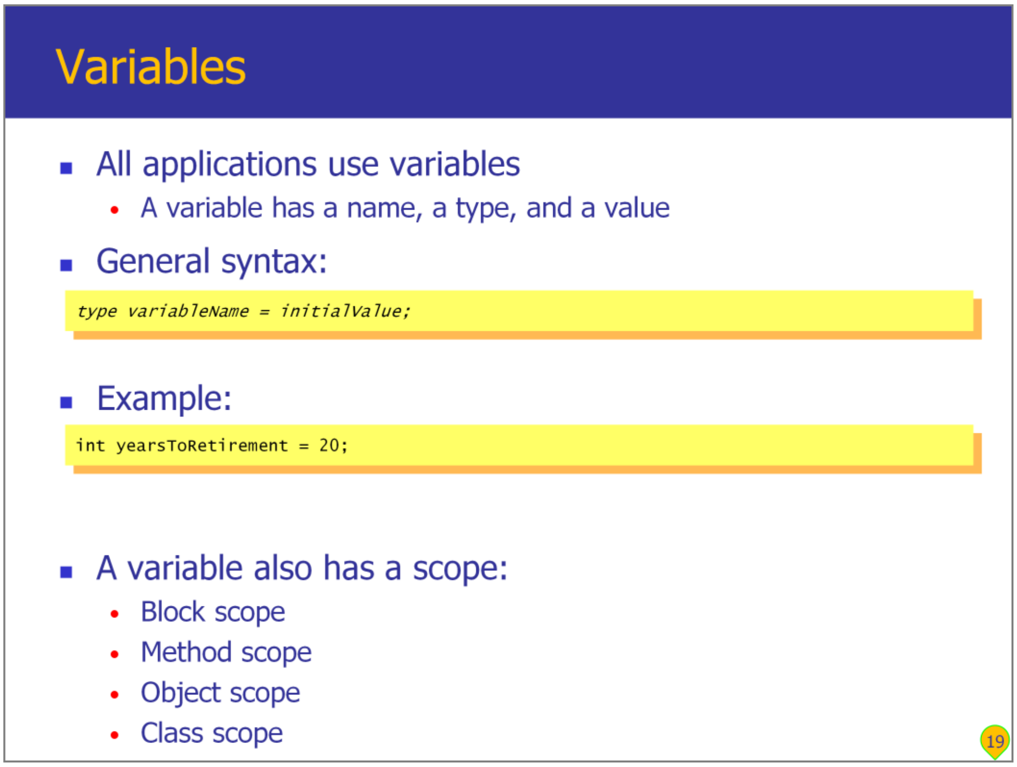
Note that the `java.lang` package is very special in Java. This package contains fundamental classes such as `System` and `String`, and is automatically imported in every Java source file – you never need to import it explicitly.



4. Declaring and Using Variables

- Variables
- Constants
- Primitive types
- Primitive type values
- Number literals in Java SE 7
- Reference types
- Using objects
- Local variables
- Instance variables and class variables

This section defines the rules for defining variables in Java. As part of this discussion, we'll explain the difference between primitive types and reference types, and we'll talk about related issues such as default values, scope, etc.



Variables

- All applications use variables
  - A variable has a name, a type, and a value
- General syntax:

```
type variableName = initialValue;
```

- Example:

```
int yearsToRetirement = 20;
```

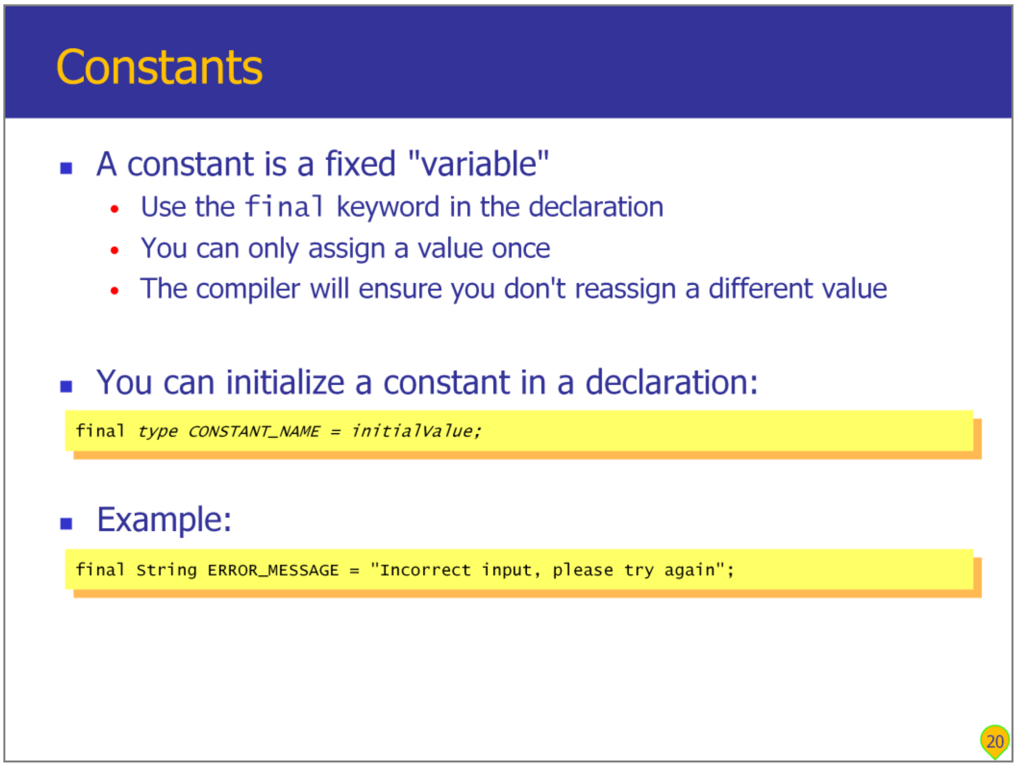
- A variable also has a scope:
  - Block scope
  - Method scope
  - Object scope
  - Class scope

19

Java is a static, strongly-typed language. This means you must declare every variable before you can use it. In the variable declaration, you specify the type of the variable and its name. Optionally, you can also specify an initial value.

Every variable has a scope:

- Block scope
  This is when you declare a variable inside `{}`, e.g. in a loop body. The variable comes into scope at the point of declaration, and goes out of scope at the `}`.
- Method scope
  This is when you declare a variable elsewhere in a method. The variable comes into scope at the point of declaration, and goes out of scope when the method finishes.
- Object scope
  This is when you declare a variable in a class (outside of any particular method). The variable comes into scope when you create an instance of the class (i.e. when you create the object). The variable can be accessed by any method in the class, regardless of where you declared the variable in the class body. The variable goes out of scope when the object dies.
- Class scope
  This is when you declare a variable in a class, using the `static` keyword. The variable comes into scope conceptually before the class is first used in your application. The variable conceptually exists for the entire lifetime of the application.



Constants

- A constant is a fixed "variable"
  - Use the `final` keyword in the declaration
  - You can only assign a value once
  - The compiler will ensure you don't reassign a different value
- You can initialize a constant in a declaration:

```
final type CONSTANT_NAME = initialValue;
```

- Example:

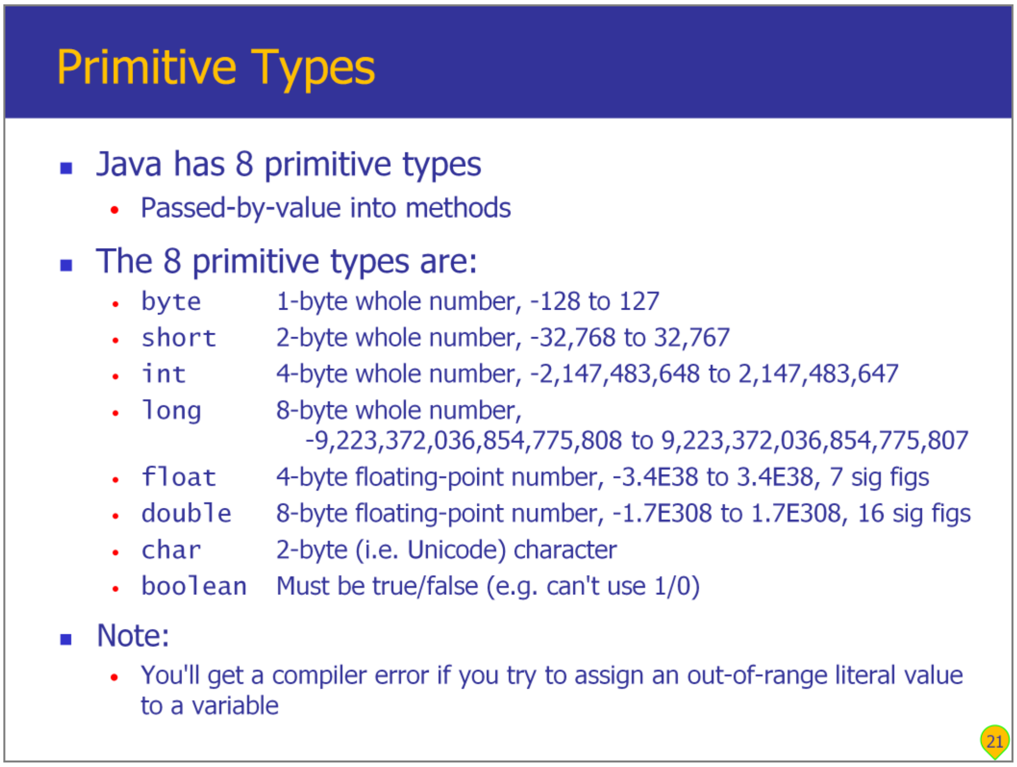
```
final String ERROR_MESSAGE = "Incorrect input, please try again";
```

20

You can use the `final` keyword when you declare a variable, to indicate that the variable will never be reassigned after it had been assigned. In other words, it's a constant. You can do this for primitive types (e.g. `int`) and for class types (e.g. `String`, `Date`, etc.)

Note the following points:

- If you try to reassign a `final` variable after it has been assigned, you'll get a compiler error.
- By convention, Java programmers tend to use upper-case for constants. This makes it obvious which variables are constant and which aren't.



Primitive Types

- Java has 8 primitive types
  - Passed-by-value into methods
- The 8 primitive types are:

`byte`	1-byte whole number, -128 to 127
`short`	2-byte whole number, -32,768 to 32,767
`int`	4-byte whole number, -2,147,483,648 to 2,147,483,647
`long`	8-byte whole number, -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4-byte floating-point number, -3.4E38 to 3.4E38, 7 sig figs
`double`	8-byte floating-point number, -1.7E308 to 1.7E308, 16 sig figs
`char`	2-byte (i.e. Unicode) character
`boolean`	Must be true/false (e.g. can't use 1/0)

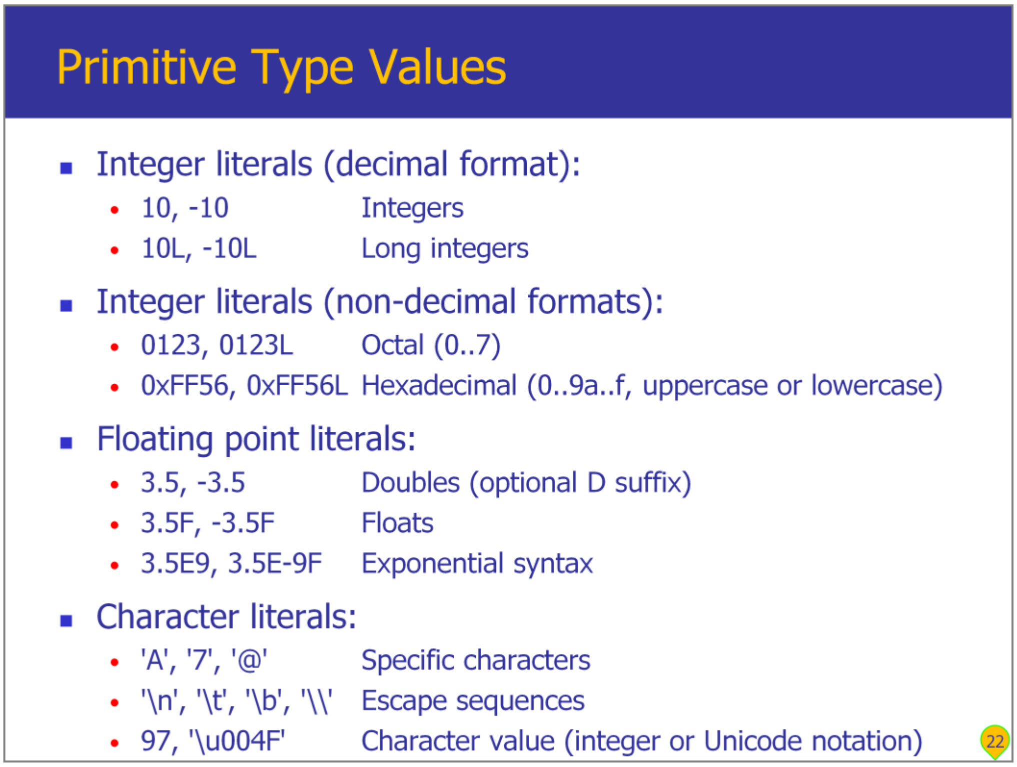
- Note:
  - You'll get a compiler error if you try to assign an out-of-range literal value to a variable

21

The Java language has just 8 primitive types, as described in the slide above. These types are hard-and-fast – they will always be the same size and offer the same range of allowable values, on every platform.

Note the following points:

- Java doesn't support unsigned data types. This is unlike C, C++, C# etc.
- When you pass primitive-type values into a method, the values are copied by value. The method receives its own local copy of the variable, which it can modify to its heart's content without affecting the original variable in the calling code.



Primitive Type Values

- Integer literals (decimal format):

10, -10	Integers
10L, -10L	Long integers

- Integer literals (non-decimal formats):

0123, 0123L	Octal (0..7)
0xFF56, 0xFF56L	Hexadecimal (0..9a..f, uppercase or lowercase)

- Floating point literals:

3.5, -3.5	Doubles (optional D suffix)
3.5F, -3.5F	Floats
3.5E9, 3.5E-9F	Exponential syntax

- Character literals:

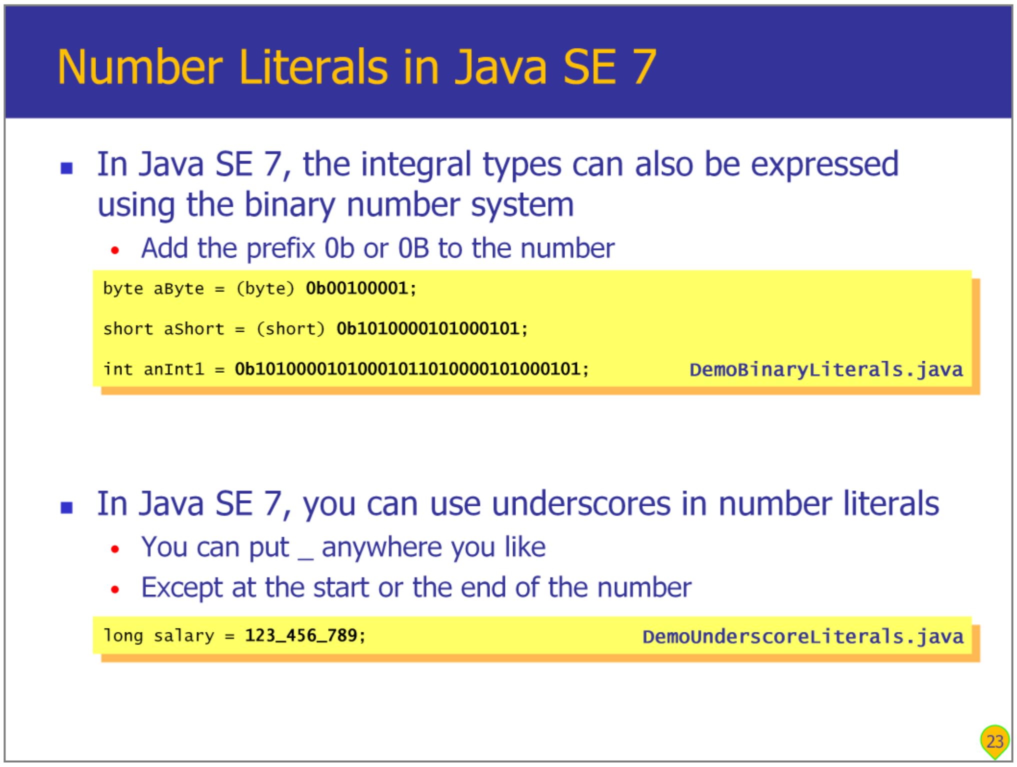
'A', '7', '@'	Specific characters
'\n', '\t', '\b', '\\'	Escape sequences
97, '\u004F'	Character value (integer or Unicode notation)

22

This slide shows how to express literal values for each primitive data type. Take a moment to review these examples, to make sure you grasp all the details.

Note the following points in particular:

- A literal fractional value such as `3.14` is a `double`, not a `float`. If you try to assign it to a `float` variable, you'll get a compiler error. The solution is to either declare the variable as a `double`, or append the letter `F` at the end of the literal value (e.g. `3.14F` or `3.14f`) to make it a `float` literal.
- Character literals are enclosed in single quotes, e.g. `'a'`. There are several special characters which use an escape-sequence syntax, e.g. `'\n'` means newline, `'\t'` means tab, etc. You can also express character literals using the appropriate Unicode character, e.g. `'\u004F'`.
- Strings literals are enclosed in double quotes, e.g. `"Wales"`. Note that `String` is a class, not a primitive type… we'll discuss it later in this chapter.



Number Literals in Java SE 7

- In Java SE 7, the integral types can also be expressed using the binary number system
  - Add the prefix 0b or 0B to the number

```
byte aByte = (byte) 0b00100001;

short aShort = (short) 0b1010000101000101;

int anInt1 = 0b10100001010001011010000101000101;        DemoBinaryLiterals.java
```

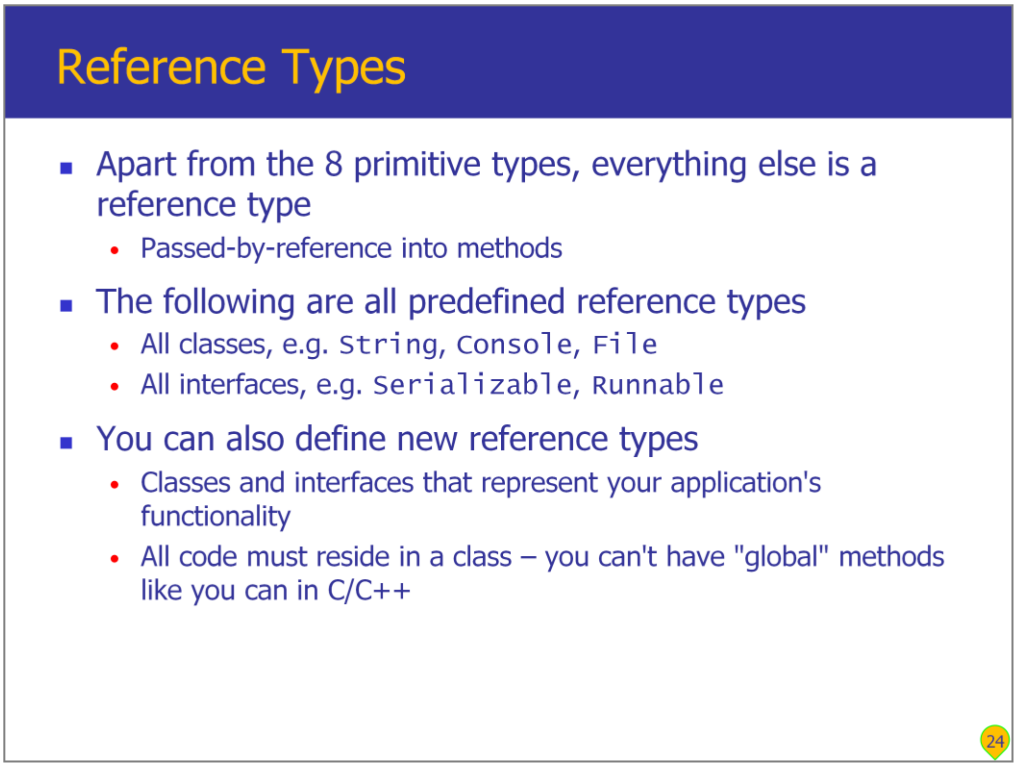
- In Java SE 7, you can use underscores in number literals
  - You can put _ anywhere you like
  - Except at the start or the end of the number

```
long salary = 123_456_789;                     DemoUnderscoreLiterals.java
```

23

Java SE 7 allows you to use the binary number system for integer literal values. This can be helpful if you're doing a lot of bit-handling, e.g. when accessing hardware registers.

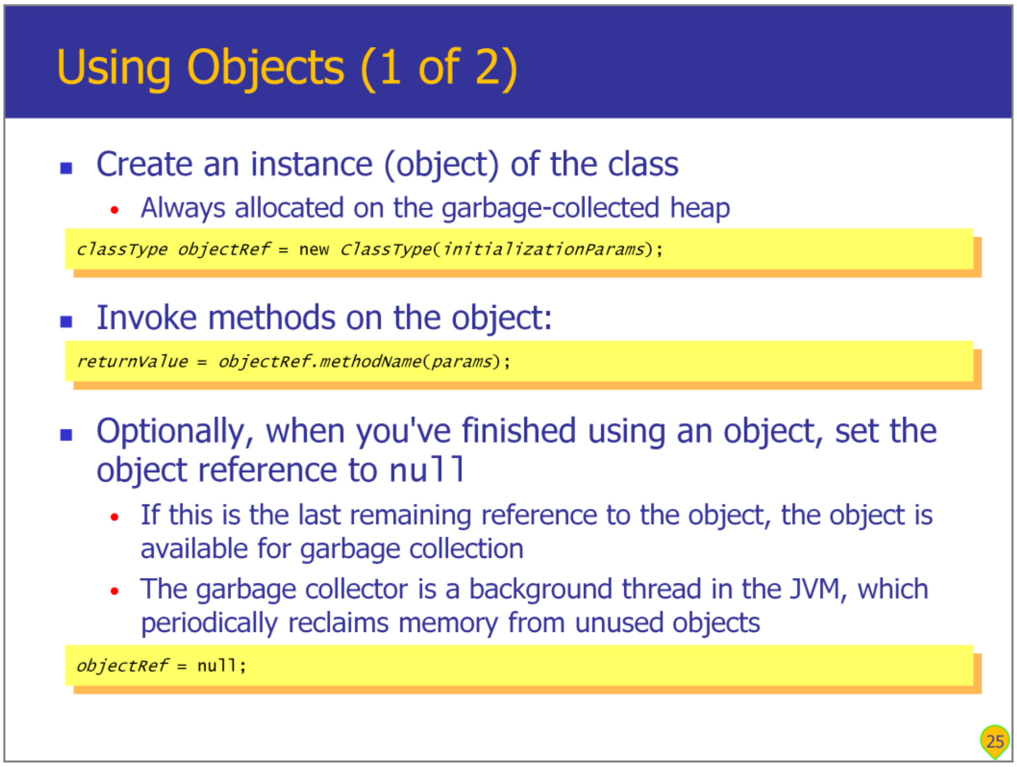
Java SE 7 also allows you to use underscores anywhere in number literals, except at the start or end of a number. This can be useful for emphasizing thousand separators in large numbers, but note you can put the _ at any location you like if you think it makes the number more readable.



Reference Types

- Apart from the 8 primitive types, everything else is a reference type
  - Passed-by-reference into methods
- The following are all predefined reference types
  - All classes, e.g. `String`, `Console`, `File`
  - All interfaces, e.g. `Serializable`, `Runnable`
- You can also define new reference types
  - Classes and interfaces that represent your application's functionality
  - All code must reside in a class – you can't have "global" methods like you can in C/C++

Apart from the 8 primitive types described on the previous page, every other type in Java is a reference type, i.e. a class or an interface. We'll discuss classes and interfaces in great detail in the rest of the course!



Using Objects (1 of 2)

- Create an instance (object) of the class
  - Always allocated on the garbage-collected heap

```
classType objectRef = new ClassType(initializationParams);
```

- Invoke methods on the object:

```
returnValue = objectRef.methodName(params);
```

- Optionally, when you've finished using an object, set the object reference to `null`
  - If this is the last remaining reference to the object, the object is available for garbage collection
  - The garbage collector is a background thread in the JVM, which periodically reclaims memory from unused objects

```
objectRef = null;
```

25

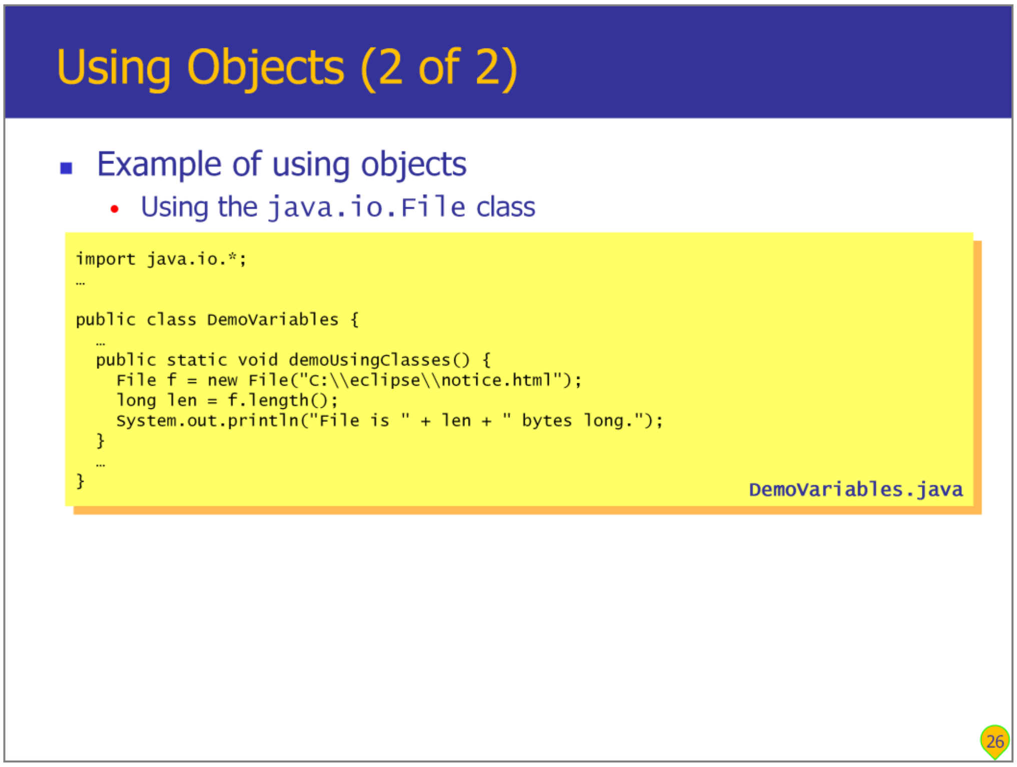
This slide gives you a foretaste of classes and objects. First, some basic terminology:

- A class is a blueprint for creating objects. A class specifies what data you want to store in an object, and what operations you want to manipulate that data. For example, a `BankAccount` class might define data members such as `balance` and `accountHolder`, plus operations (or "methods") such as `deposit()`, `withdraw()`, etc.
- An object is an instance of a class. It occupies memory and holds its own set data for that particular object.

The first code box in the slide shows how to create an object, via the `new` operator. When you use `new`, it allocates memory for the object and returns a pointer (or a "reference") to this memory. You store this reference in a variable, so you can get access it again later – we call this variable an "object reference".

The second code box in the slide shows how to access members on an object. You use the object reference, followed by a dot, followed by the member you want to access. Normally, the only members you can access in an object are its methods (because the data members are typically private).

The third code box in the slide shows how to set the object reference to `null` when you no longer need the object. The JVM has a garbage collector component that periodically sweeps up objects that are no longer referenced anywhere in your application.



Using Objects (2 of 2)

- Example of using objects
  - Using the `java.io.File` class

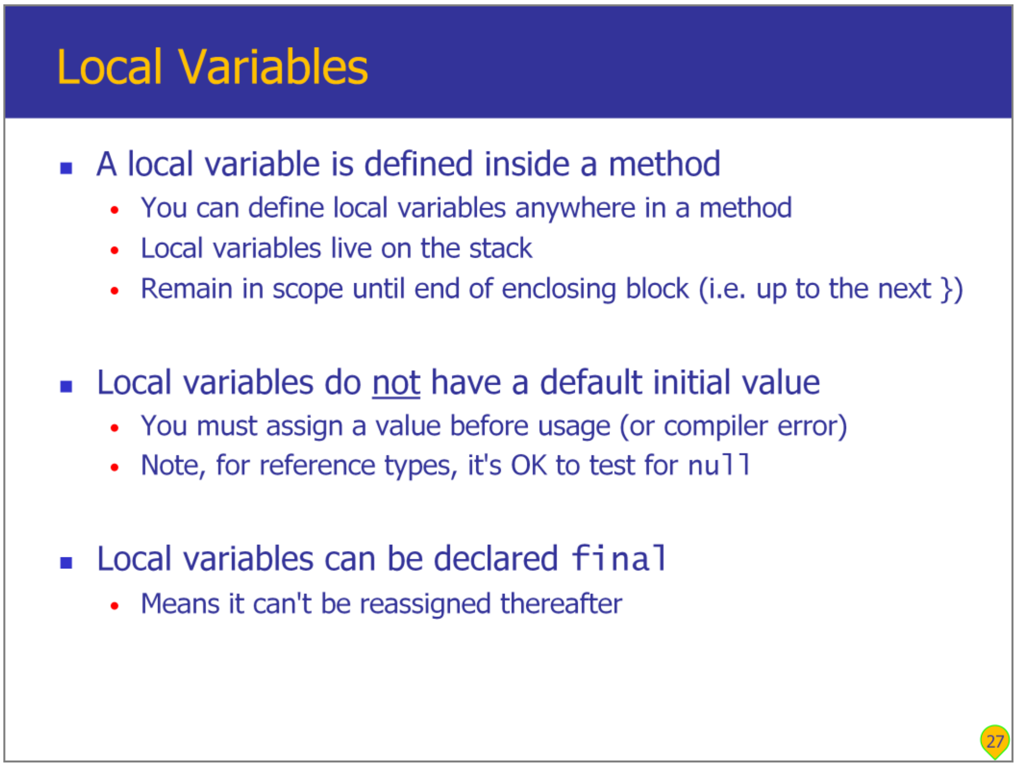
```
import java.io.*;
…

public class DemoVariables {
  …
  public static void demoUsingClasses() {
    File f = new File("C:\\eclipse\\notice.html");
    long len = f.length();
    System.out.println("File is " + len + " bytes long.");
  }
  …
}
```

`DemoVariables.java`

26

This slide shows a simple but complete example of how to create and use objects in Java. This example is available in the `DemoVariables.java` source code file in the demo project for this chapter.

Try running this example in Eclipse, to see how it works.



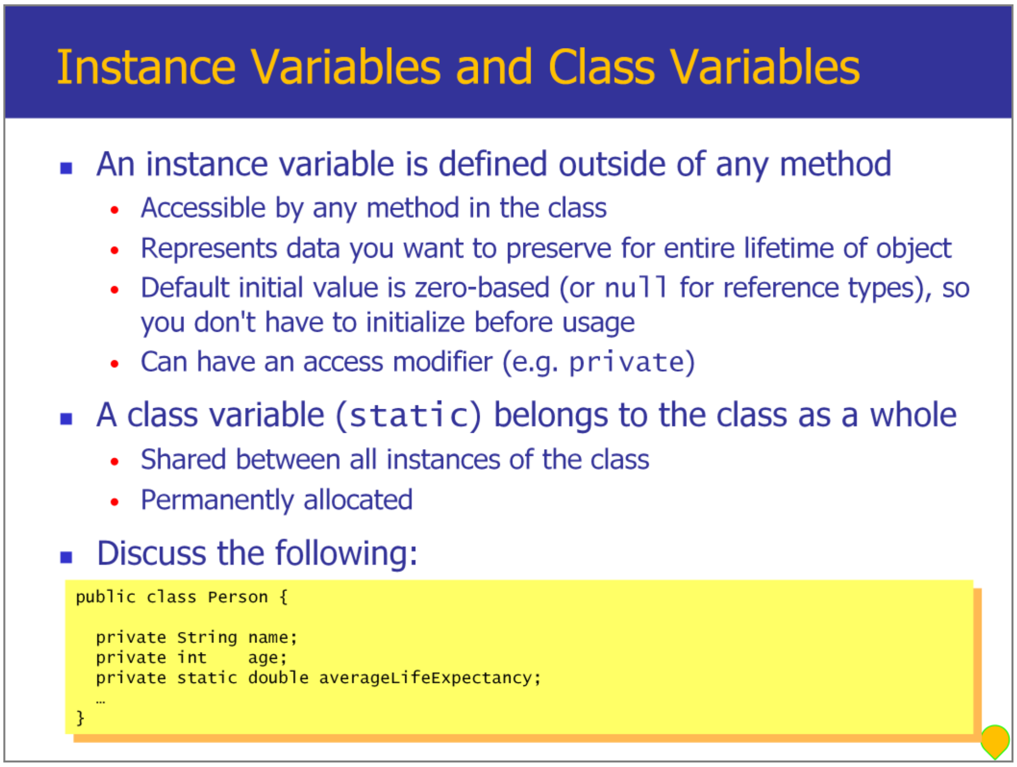
Local Variables

- A local variable is defined inside a method
  - You can define local variables anywhere in a method
  - Local variables live on the stack
  - Remain in scope until end of enclosing block (i.e. up to the next })
- Local variables do <u>not</u> have a default initial value
  - You must assign a value before usage (or compiler error)
  - Note, for reference types, it's OK to test for `null`
- Local variables can be declared `final`
  - Means it can't be reassigned thereafter

27

Java has some rather surprising rules regarding variable initialization. Basically there are two different rules, depending on where you declare the variable – i.e. local or non-local.

This slide describes what happens if you declare a local variable, i.e. somewhere inside a method. The variable does not have a default initial value (not even `0`, or `null`, or anything). Furthermore, if you try to use the variable before you have explicitly initialized it, you'll get a compiler error.



Instance Variables and Class Variables

- An instance variable is defined outside of any method
  - Accessible by any method in the class
  - Represents data you want to preserve for entire lifetime of object
  - Default initial value is zero-based (or `null` for reference types), so you don't have to initialize before usage
  - Can have an access modifier (e.g. `private`)
- A class variable (`static`) belongs to the class as a whole
  - Shared between all instances of the class
  - Permanently allocated
- Discuss the following:

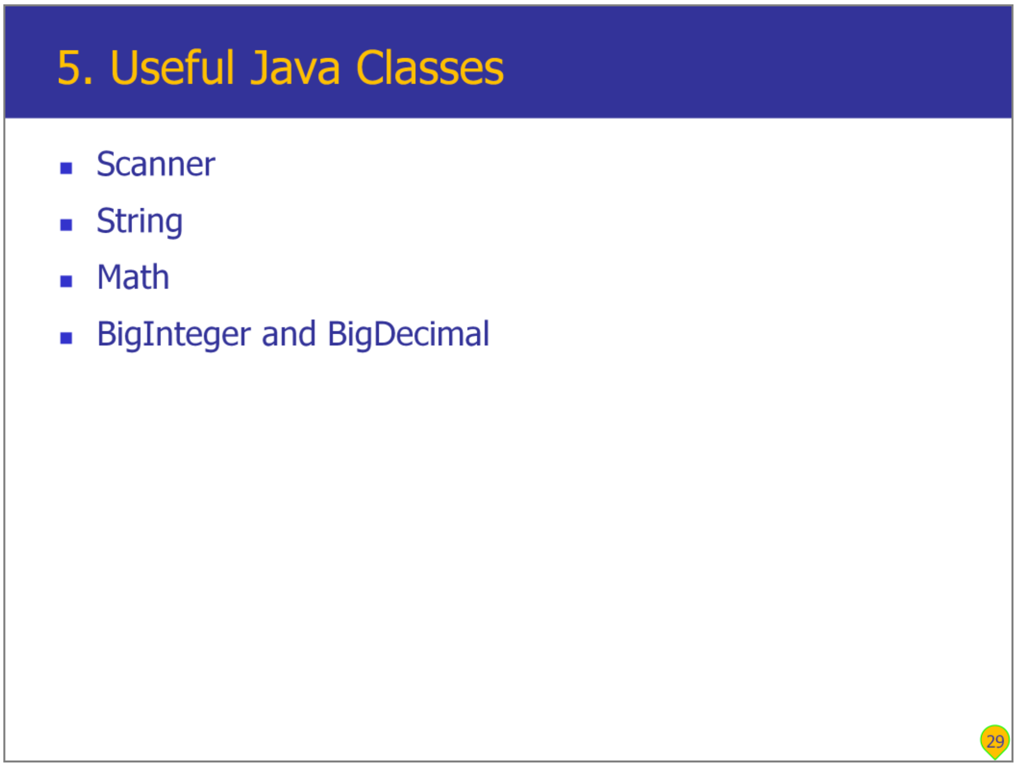
```
public class Person {

  private String name;
  private int    age;
  private static double averageLifeExpectancy;
  …
}
```

If you declare a variable anywhere outside a method (i.e. object or class scope), the variable has a default initial value based on its type:

- Integral variables (`byte`, `short`, `int`, `long`) are initialized to 0 by default.
- Floating-point variables (`float`, `double`) are initialized to `0.0` by default.
- Character variables are initialized to `'\u0000'` by default.
- Boolean variables are initialized to `false` by default.
- All non-primitive variables are initialized to `null` by default.

The example in the slide shows how to define a simple `Person` class with some data members. Note the following points:

- `name` and `age` are instance variables. Every `Person` instance (i.e. object) will have its own values for these data members
- `averageLifeExpectancy` a class variable – that's the effect of the `static` keyword. There will only be a single copy of `averageLifeExpectancy`, which will be shared by all `Person` instances. It's a bit like a global variable, which is limited in scope to the `Person` class.



5. Useful Java Classes

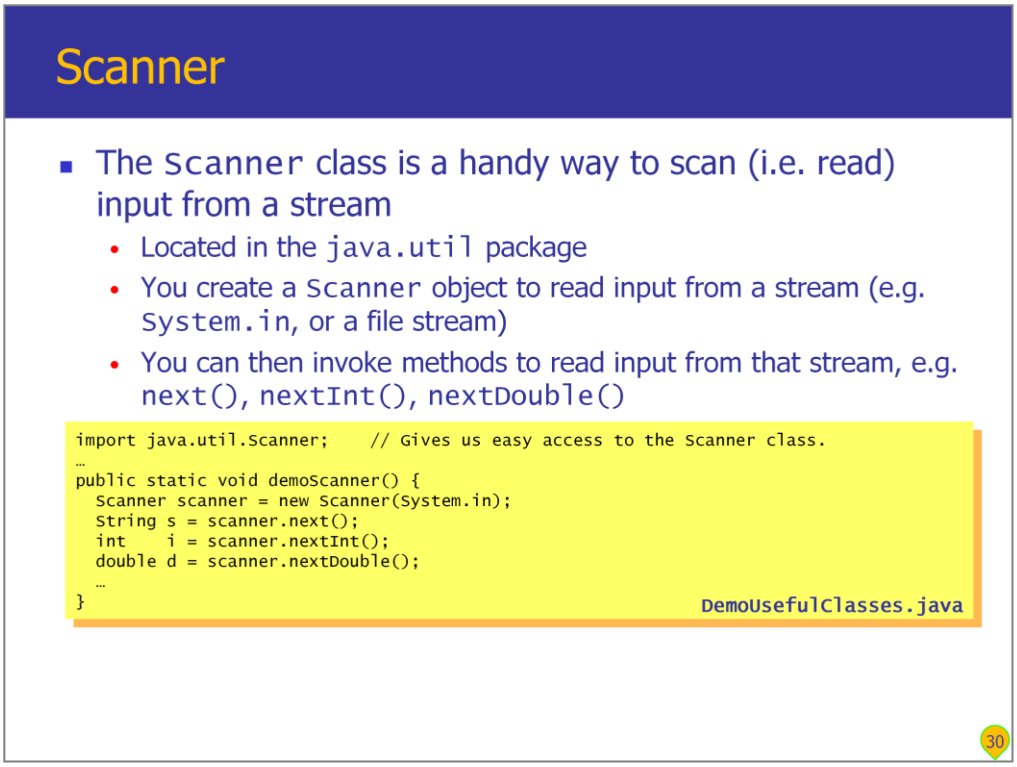
- Scanner
- String
- Math
- BigInteger and BigDecimal

29

The Java SE library contains thousands of classes. Getting to grips with these classes is a gradual process – bit by bit, you'll discover the classes you need and dig into the online documentation to see how to use them.

To get the ball rolling, this section introduces a few of the simple classes:

- `Scanner`, for doing console I/O.
- `String`, for holding and manipulating text.
- `Math`, for performing mathematical operations such as sin and cos.
- `BigInteger` and `BigDecimal`, for representing and manipulating very large whole numbers and fractions respectively.



Scanner

- The `Scanner` class is a handy way to scan (i.e. read) input from a stream
  - Located in the `java.util` package
  - You create a `Scanner` object to read input from a stream (e.g. `System.in`, or a file stream)
  - You can then invoke methods to read input from that stream, e.g. `next()`, `nextInt()`, `nextDouble()`

```
import java.util.Scanner;    // Gives us easy access to the Scanner class.
…
public static void demoScanner() {
  Scanner scanner = new Scanner(System.in);
  String s = scanner.next();
  int    i = scanner.nextInt();
  double d = scanner.nextDouble();
  …
}
```

`DemoUsefulClasses.java`

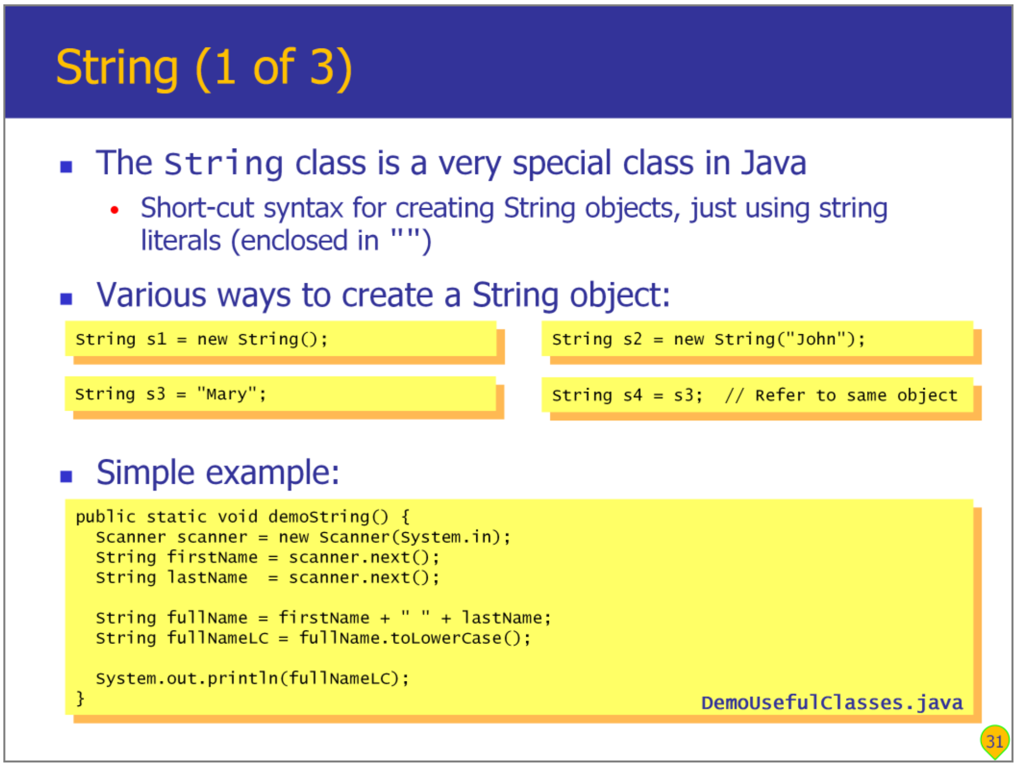
This example shows how to use the `Scanner` class, which is defined in the `java.util` package. The `Scanner` class allows you to read formatted input from the console (or another stream, e.g. a file).

For the full code for this example, see the `demoScanner()` method in the `DemoUsefulClasses.java` demo file. Try running the example in Eclipse, to see how it works.

For full online information about the `Scanner` class, see the following web site:

- http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

There's full JavaDoc documentation available online for all classes in the Java SE and Java EE libraries, organized by package. Learning how to find your way around JavaDoc documentation is an important skill for Java programmers.



String (1 of 3)

- The `String` class is a very special class in Java
  - Short-cut syntax for creating String objects, just using string literals (enclosed in "")
- Various ways to create a String object:

```
String s1 = new String();
```

```
String s2 = new String("John");
```

```
String s3 = "Mary";
```

```
String s4 = s3;  // Refer to same object
```

- Simple example:

```
public static void demoString() {
  Scanner scanner = new Scanner(System.in);
  String firstName = scanner.next();
  String lastName  = scanner.next();

  String fullName = firstName + " " + lastName;
  String fullNameLC = fullName.toLowerCase();

  System.out.println(fullNameLC);
}
```

DemoUsefulClasses.java

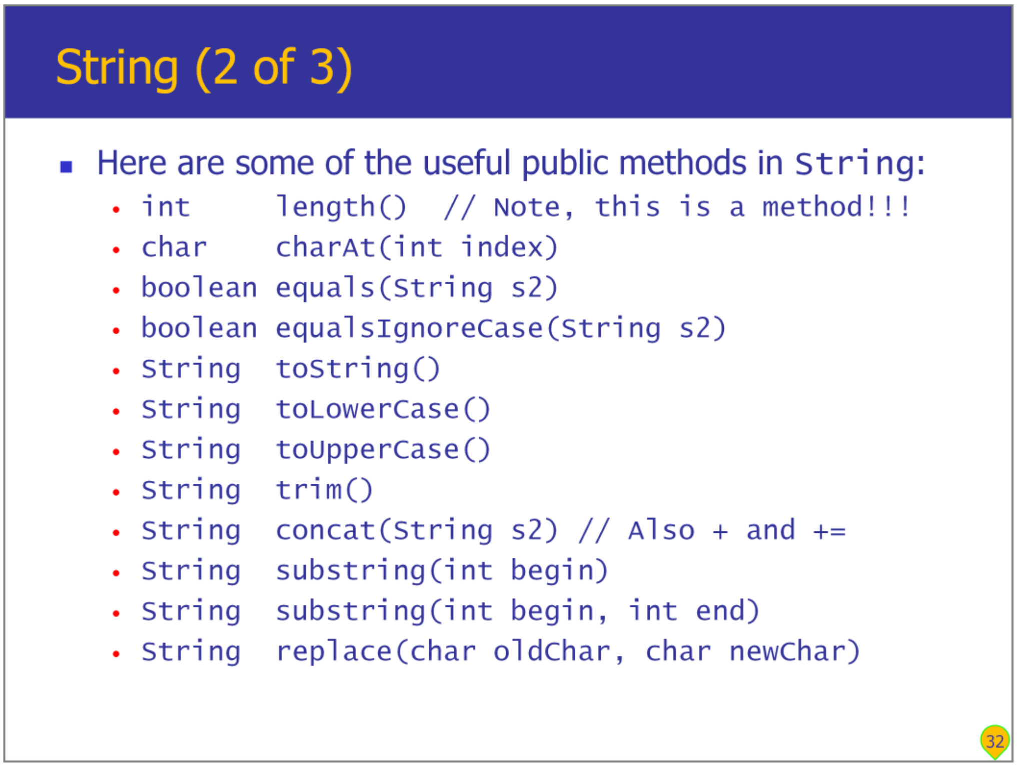
31

The `String` class is defined in the `java.lang` package, and allows you to store and manipulate textual content in your application.

The simplest way to create a `String` object is via the "Mary" example in the slide above. In this example, we can say that s3 is a variable that refers to a `String` object, and the `String` object contains the text "Mary".

Also note the s4 example in the slide. In this example, s4 points to the same String object as does s3. If you know about pointers in C or C++, there's a similar thing going on here.

For lower code box shows some simple string manipulation. You can run this example in Eclipse as well, to see how it works.



String (2 of 3)

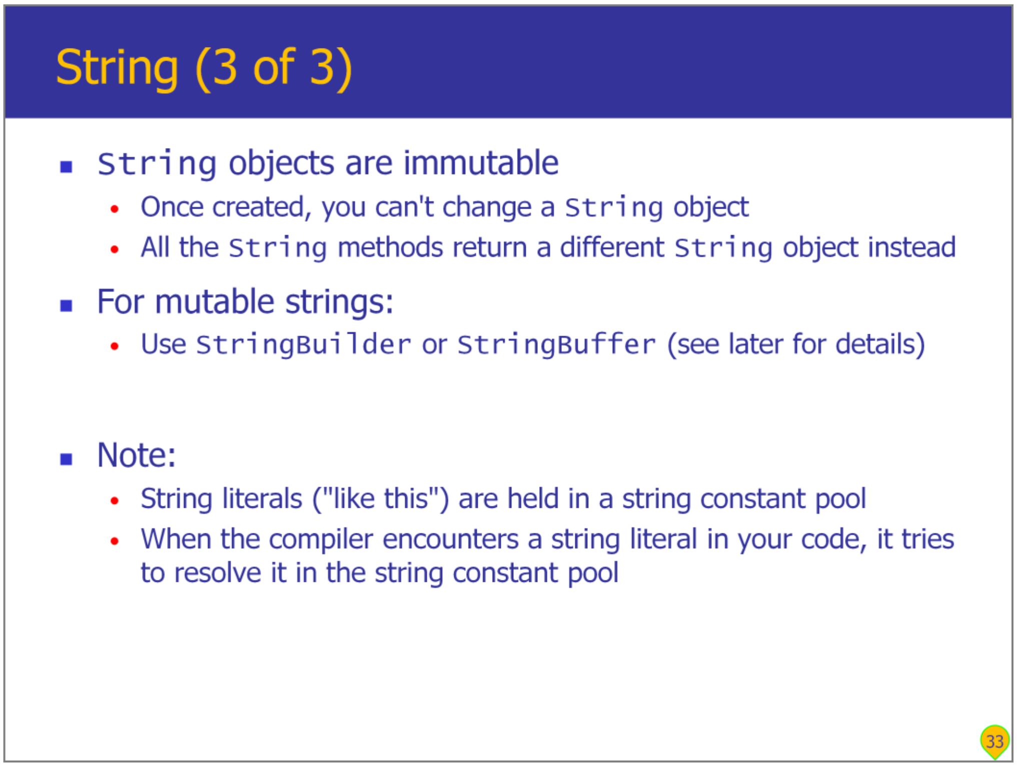
- Here are some of the useful public methods in `String`:

```
• int     length()  // Note, this is a method!!!
• char    charAt(int index)
• boolean equals(String s2)
• boolean equalsIgnoreCase(String s2)
• String  toString()
• String  toLowerCase()
• String  toUpperCase()
• String  trim()
• String  concat(String s2) // Also + and +=
• String  substring(int begin)
• String  substring(int begin, int end)
• String  replace(char oldChar, char newChar)
```

32

This slide lists some of the most common methods in the `String` class. For full online information about these and other methods in `String`, see the following web site:

- http://docs.oracle.com/javase/7/docs/api/java/lang/String.html



String (3 of 3)

- `String` objects are immutable
  - Once created, you can't change a `String` object
  - All the `String` methods return a different `String` object instead
- For mutable strings:
  - Use `StringBuilder` or `StringBuffer` (see later for details)
- Note:
  - String literals ("like this") are held in a string constant pool
  - When the compiler encounters a string literal in your code, it tries to resolve it in the string constant pool

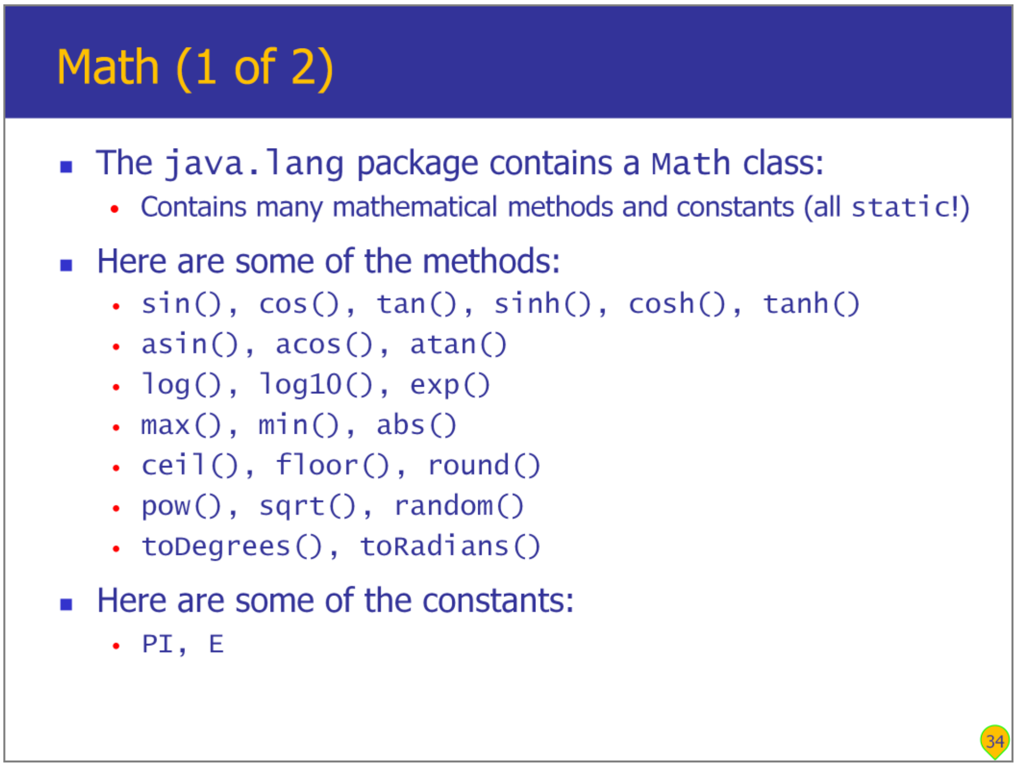
33

Note that the `String` class is immutable. In other words, once you've created a `String` object, there's nothing you can do to change its contents.

This might seem a bit odd at first glance, because the `String` class does seem to offer methods for changing a string – e.g. `toUpperCase()`, `toLowerCase()`, `trim()`, etc. In reality, these methods don't actually change the original string value, but instead create and return a new `String` object with the appropriate content.

For example, if you want to convert a string value to upper-case, you must write code such as the following:

```
String country = "usa";
country = country.toUpperCase();
```



Math (1 of 2)

- The `java.lang` package contains a `Math` class:
  - Contains many mathematical methods and constants (all `static`!)
- Here are some of the methods:
  - `sin(), cos(), tan(), sinh(), cosh(), tanh()`
  - `asin(), acos(), atan()`
  - `log(), log10(), exp()`
  - `max(), min(), abs()`
  - `ceil(), floor(), round()`
  - `pow(), sqrt(), random()`
  - `toDegrees(), toRadians()`
- Here are some of the constants:
  - `PI, E`

34

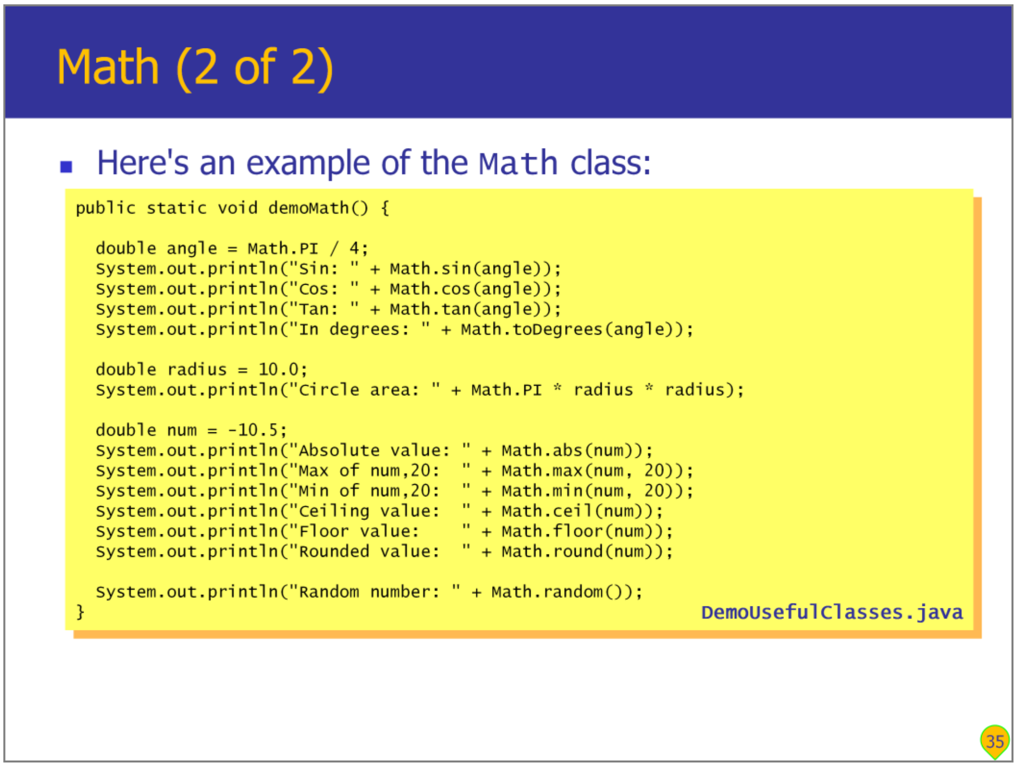
The `Math` class is defined in the `java.lang` package, and contains a host of methods for performing mathematical operations. All of these methods are `static`, which means you invoke the methods via the name of the class (i.e. `Math`). For example:

```
    double result = Math.sin(anAngleInRadians);
```

We'll describe exactly what `static` methods are, and why they're useful, later in the course.

For full online information about the `Math` class see the following web site:

- http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html



Math (2 of 2)

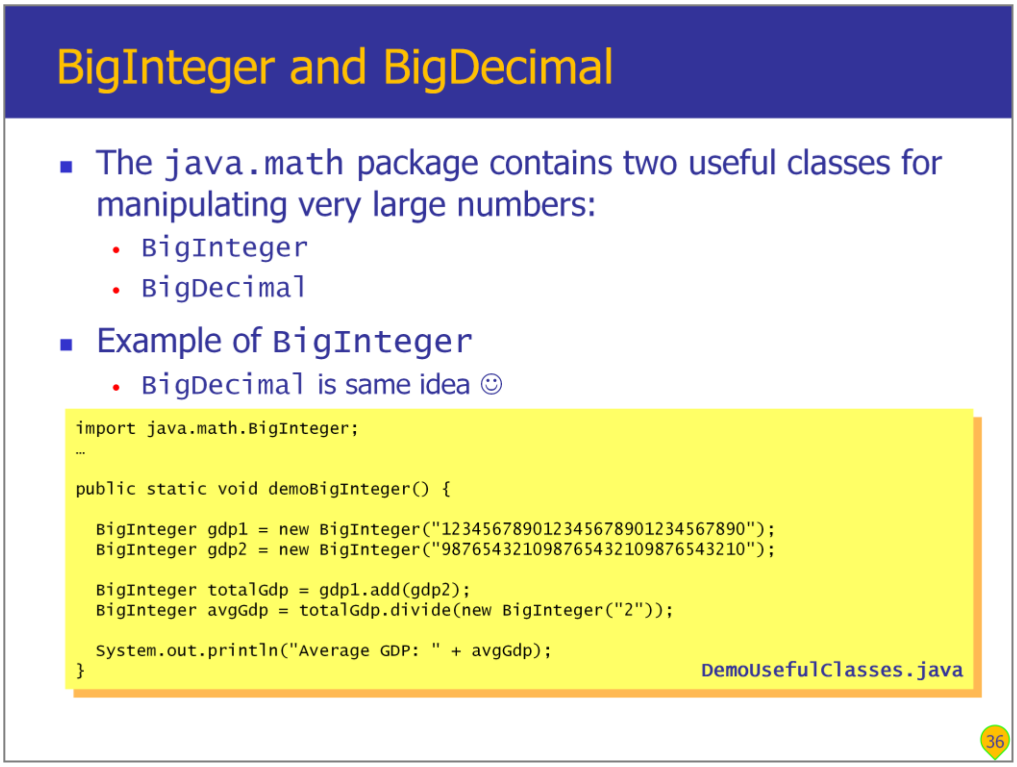
- Here's an example of the `Math` class:

```
public static void demoMath() {

  double angle = Math.PI / 4;
  System.out.println("Sin: " + Math.sin(angle));
  System.out.println("Cos: " + Math.cos(angle));
  System.out.println("Tan: " + Math.tan(angle));
  System.out.println("In degrees: " + Math.toDegrees(angle));

  double radius = 10.0;
  System.out.println("Circle area: " + Math.PI * radius * radius);

  double num = -10.5;
  System.out.println("Absolute value: " + Math.abs(num));
  System.out.println("Max of num,20:  " + Math.max(num, 20));
  System.out.println("Min of num,20:  " + Math.min(num, 20));
  System.out.println("Ceiling value:  " + Math.ceil(num));
  System.out.println("Floor value:    " + Math.floor(num));
  System.out.println("Rounded value:  " + Math.round(num));

  System.out.println("Random number: " + Math.random());
}
```

`DemoUsefulClasses.java`

35

This example shows how to use various methods in the `Math` class. For the full code for this example, see `demoMath()` in `DemoUsefulClasses.java`.



BigInteger and BigDecimal

- The `java.math` package contains two useful classes for manipulating very large numbers:
  - `BigInteger`
  - `BigDecimal`
- Example of `BigInteger`
  - `BigDecimal` is same idea ☺

```
import java.math.BigInteger;
…

public static void demoBigInteger() {

  BigInteger gdp1 = new BigInteger("123456789012345678901234567890");
  BigInteger gdp2 = new BigInteger("987654321098765432109876543210");

  BigInteger totalGdp = gdp1.add(gdp2);
  BigInteger avgGdp = totalGdp.divide(new BigInteger("2"));

  System.out.println("Average GDP: " + avgGdp);
}
```

DemoUsefulClasses.java

36

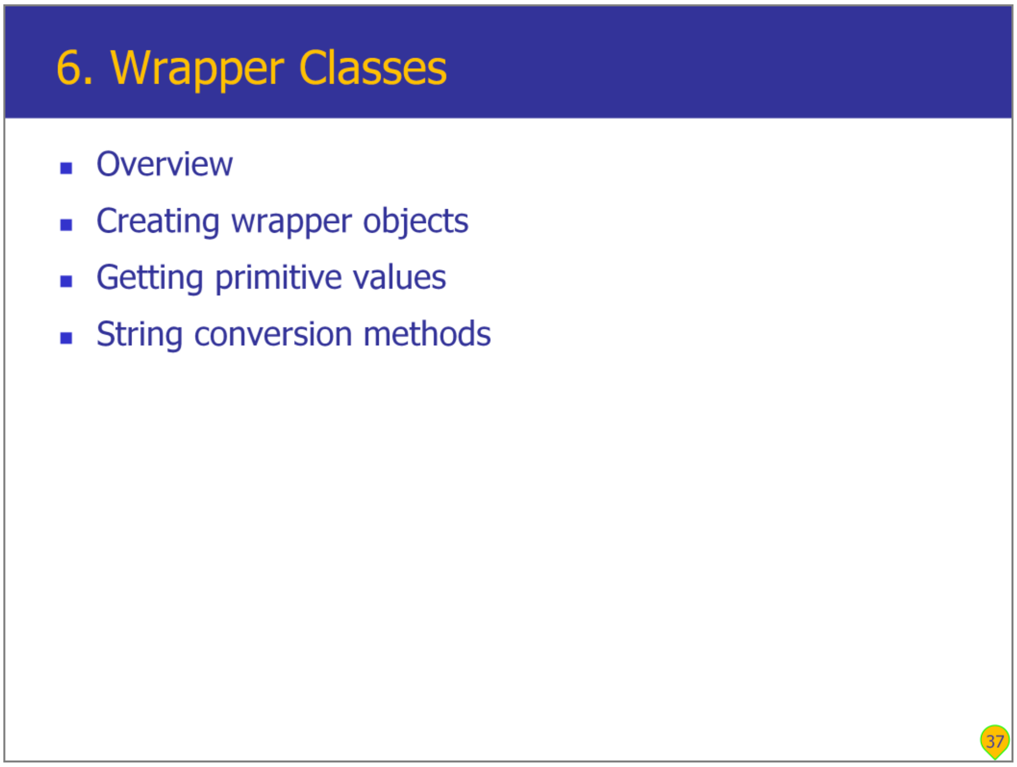
To conclude this section on commonplace Java classes, we present the `BigInteger` and `BigDecimal` classes. These classes are useful if you need to store extremely large values with a high degree of accuracy. Both of these classes are defined in the `java.math` package.

One word of caution: these are classes, not primitive types, so you can't manipulate values using simple operators such as + and `-`. Instead, you must invoke methods such as `add()` and `subtract()`. Also, these classes are hideously slow compared to primitive types, so use them sparingly.

The example in the slide shows how to use `BigInteger` (`BigDecimal` is very similar). For the full code for this example, see `demoBigInteger()` in `DemoUsefulClasses.java`.

For full online information about the `BigInteger` and `BigDecimal` classes, see the following web sites:

- http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html
- http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html

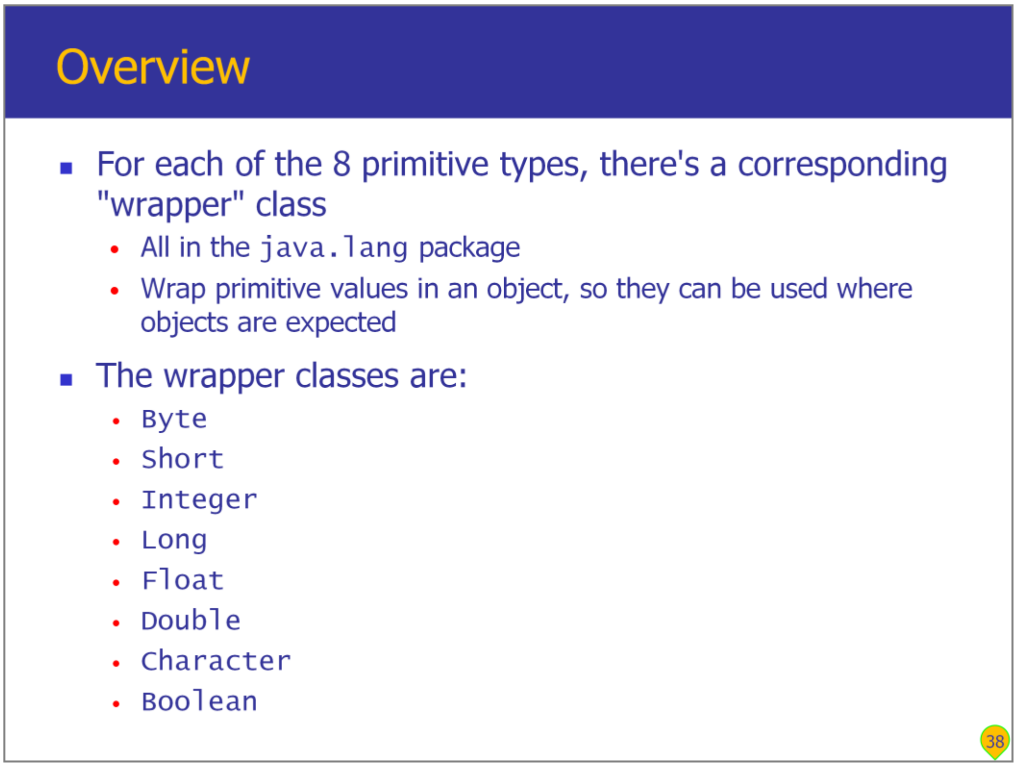


6. Wrapper Classes

- Overview
- Creating wrapper objects
- Getting primitive values
- String conversion methods

37

This is the final section in this chapter, and introduces you to the weird and wonderful world of wrapper classes.



Overview

- For each of the 8 primitive types, there's a corresponding "wrapper" class
  - All in the `java.lang` package
  - Wrap primitive values in an object, so they can be used where objects are expected
- The wrapper classes are:
  - `Byte`
  - `Short`
  - `Integer`
  - `Long`
  - `Float`
  - `Double`
  - `Character`
  - `Boolean`

38

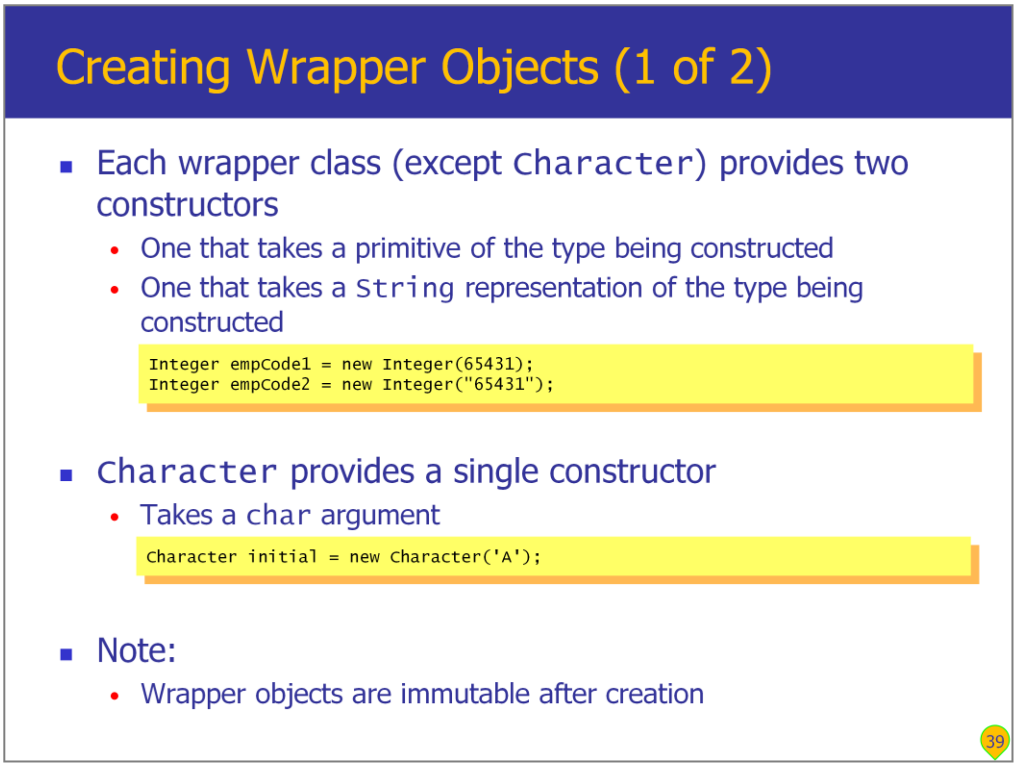
For each of the primitive types, there is a corresponding wrapper class that houses useful methods for manipulating the relevant primitive type (e.g. converting to and from strings, determining the minimum and maximum values for that type, etc.).

Here's how the wrapper class relates to the corresponding primitive type:

- `Byte` defines useful methods for manipulating `byte` values.
- `Short` defines useful methods for manipulating `short` values.
- `Integer` defines useful methods for manipulating `int` values.
- etc…

All of the wrapper classes are defined in the `java.lang` package. For more information about the wrapper classes, see the following web sites:

- http://docs.oracle.com/javase/7/docs/api/java/lang/Byte.html
- http://docs.oracle.com/javase/7/docs/api/java/lang/Short.html
- http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html
- etc…



Creating Wrapper Objects (1 of 2)

- Each wrapper class (except `Character`) provides two constructors
  - One that takes a primitive of the type being constructed
  - One that takes a `String` representation of the type being constructed

```
Integer empCode1 = new Integer(65431);
Integer empCode2 = new Integer("65431");
```

- `Character` provides a single constructor
  - Takes a `char` argument

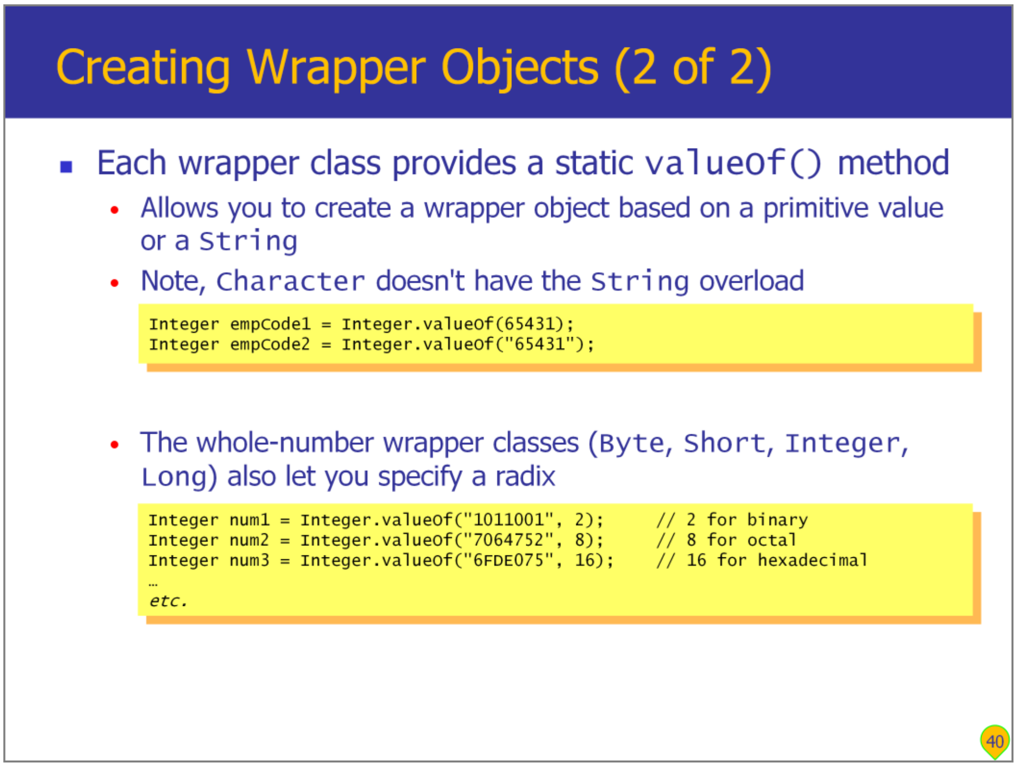
```
Character initial = new Character('A');
```

- Note:
  - Wrapper objects are immutable after creation

39

The first code block in the slide shows the general way for creating a wrapper object, based on a simple value. The first statement creates an `Integer` object from an `int` value, and the second statement creates an `Integer` object from a string.

The second code block shows how to create a `Character` object. The only option here is to supply a character value, such as `'A'` in the example.



Creating Wrapper Objects (2 of 2)

- Each wrapper class provides a static `valueOf()` method
  - Allows you to create a wrapper object based on a primitive value or a `String`
  - Note, `Character` doesn't have the `String` overload

```
Integer empCode1 = Integer.valueOf(65431);
Integer empCode2 = Integer.valueOf("65431");
```

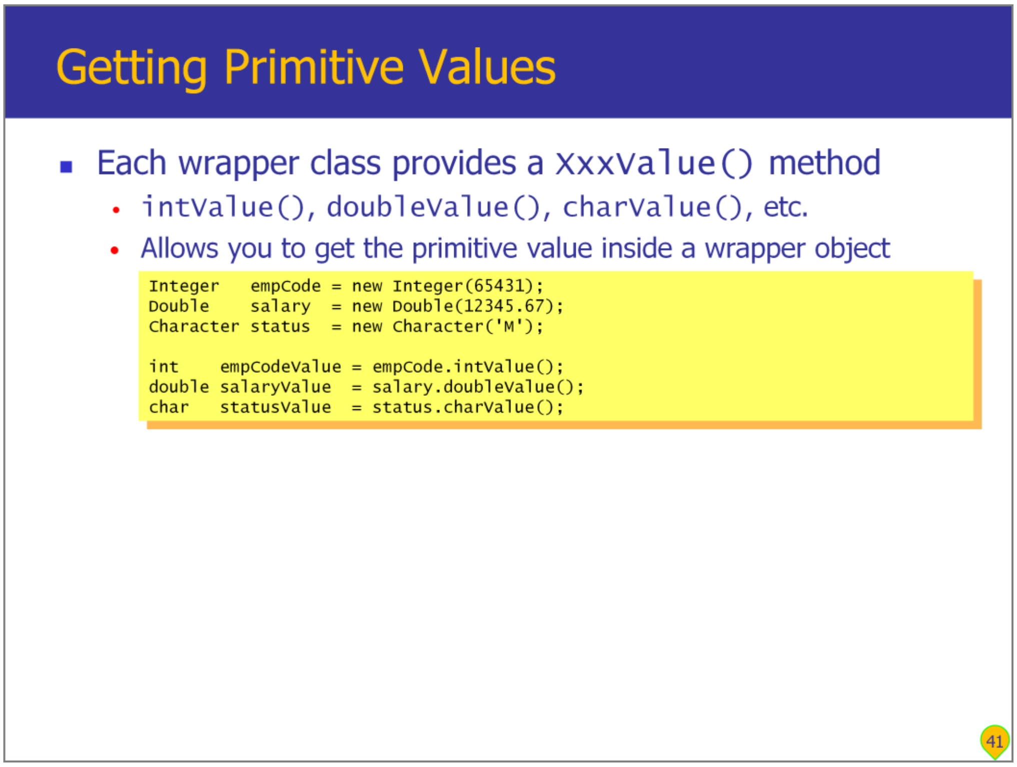
  - The whole-number wrapper classes (`Byte`, `Short`, `Integer`, `Long`) also let you specify a radix

```
Integer num1 = Integer.valueOf("1011001", 2);     // 2 for binary
Integer num2 = Integer.valueOf("7064752", 8);     // 8 for octal
Integer num3 = Integer.valueOf("6FDE075", 16);    // 16 for hexadecimal
…
etc.
```

40

All of the wrapper classes have a `static` method named `valueOf()`. You can call these methods to create a wrapper object (e.g. an `Integer` object) based on simple value or a string.

The lower code box shows some of the flexibility of these methods – for example you can pass a radix parameter into the `Integer.valueOf()` method indicating whether to interpret the value as binary, octal, or hexadecimal.



Getting Primitive Values

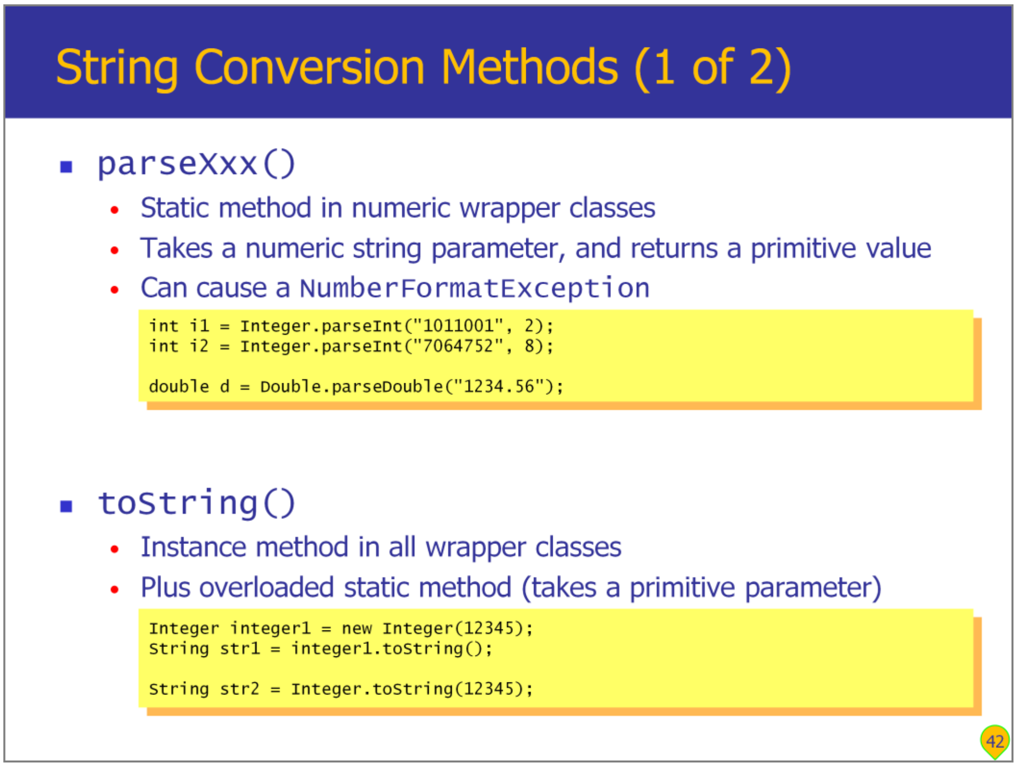
- Each wrapper class provides a `XxxValue()` method
  - `intValue()`, `doubleValue()`, `charValue()`, etc.
  - Allows you to get the primitive value inside a wrapper object

```
Integer   empCode = new Integer(65431);
Double    salary  = new Double(12345.67);
Character status  = new Character('M');

int    empCodeValue = empCode.intValue();
double salaryValue  = salary.doubleValue();
char   statusValue  = status.charValue();
```

41

All of the wrapper classes have a method named `xxxValue()`, which convert a wrapper object (e.g. an `Integer` object) into the appropriate primitive-type value (e.g. `int`).

The actual names of these methods are `intValue()`, `doubleValue()`, `charValue()`, and so on.



String Conversion Methods (1 of 2)

- `parseXxx()`
  - Static method in numeric wrapper classes
  - Takes a numeric string parameter, and returns a primitive value
  - Can cause a `NumberFormatException`

```
int i1 = Integer.parseInt("1011001", 2);
int i2 = Integer.parseInt("7064752", 8);

double d = Double.parseDouble("1234.56");
```

- `toString()`
  - Instance method in all wrapper classes
  - Plus overloaded static method (takes a primitive parameter)

```
Integer integer1 = new Integer(12345);
String str1 = integer1.toString();

String str2 = Integer.toString(12345);
```

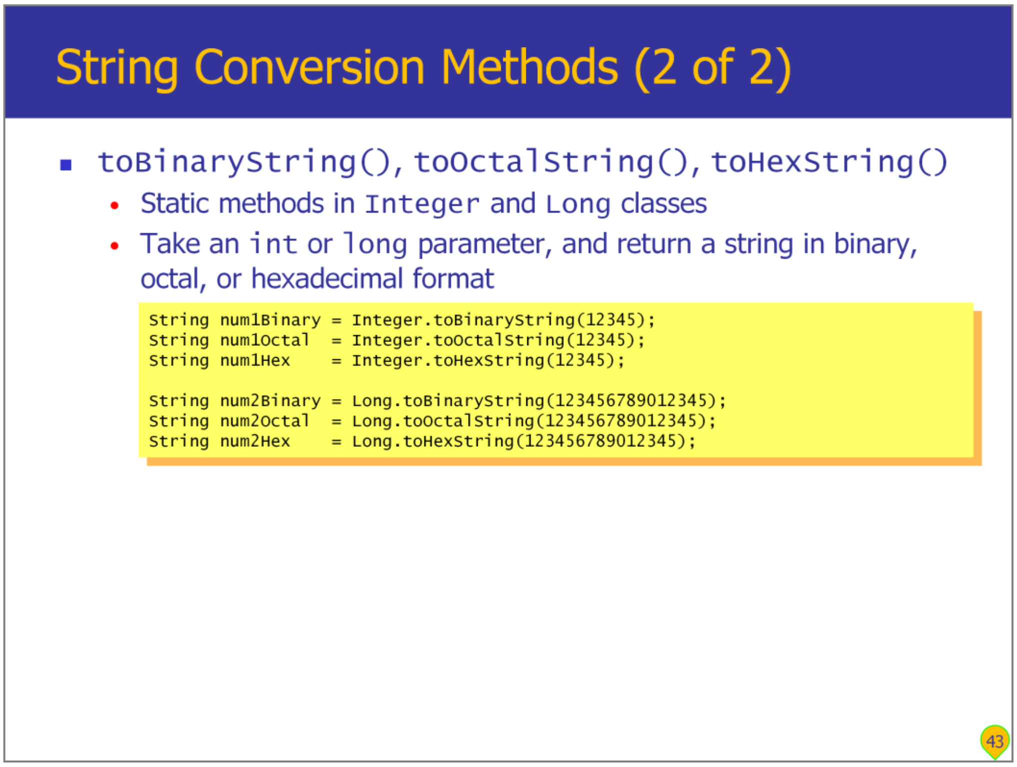
42

All of the wrapper classes have a `static` method named `parseXxx()`, which parses a string and returns an appropriate primitive-type value. For example:

- `Integer.parseInt()` parses a string value that contains a whole number, and returns that number as an `int`.
- `Double.parseDouble()` parses a string value that contains a fractional number, and returns that number as a `double`.
- etc…

All of the wrapper classes also have a method named `toString()`, which converts a simple value back into a string. There are two ways to invoke the `toString()` method:

```
// You can call toString() on a wrapper object:
String str1 = anIntegerObj.toString();

// You can call toString() directly on a wrapper class:
String str2 = Integer.toString(anIntValue);
```



String Conversion Methods (2 of 2)

- `toBinaryString()`, `toOctalString()`, `toHexString()`
  - Static methods in `Integer` and `Long` classes
  - Take an `int` or `long` parameter, and return a string in binary, octal, or hexadecimal format

```
String num1Binary = Integer.toBinaryString(12345);
String num1Octal  = Integer.toOctalString(12345);
String num1Hex    = Integer.toHexString(12345);

String num2Binary = Long.toBinaryString(123456789012345);
String num2Octal  = Long.toOctalString(123456789012345);
String num2Hex    = Long.toHexString(123456789012345);
```

43

To conclude this section, we mention a few useful methods in the `Integer` and `Long` classes that allow you to convert numbers to and from strings, using a binary, octal, or hexadecimal radix.

Any Questions?