Java SE 7 Fundamentals

Student Guide • Volume 2

D67234GC10
Edition 1.0
September 2011
D74171

ORACLE

Authors

Jill Moritz

Kenneth Somerville

Cindy Church

Technical Contributors and Reviewers

Mike Williams

Tom McGinn

Matt Heimer

Joe Darcy

Brian Goetz

Alex Buckley

Adam Messenger

Steve Watts

Editors

Smita Kommini

Richard Wallis

Graphic Designers

Rajiv Chandrabhanu

Maheshwari Krishnamurthy

Publishers

Revathi Ramamoorthy

Jayanthy Keshavamurthy

Contents

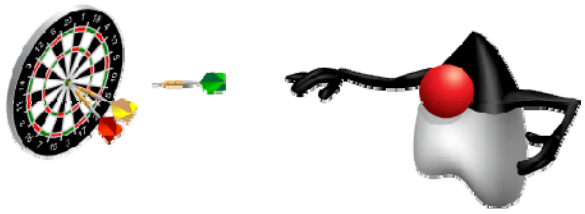
8

Creating and Using Arrays

Objectives

After completing this lesson, you should be able to:

- Declare, instantiate, and initialize a one-dimensional array
- Declare, instantiate, and initialize a two-dimensional array
- Access a value within an array
- Describe how arrays are stored in memory
- Declare and initialize an ArrayList
- Use an `args` array



ORACLE

Topics

- Overview of arrays
- Declaring, instantiating, and initializing arrays
- Accessing command-line arguments
- Working with two-dimensional arrays
- Working with ArrayList

Introduction to Arrays

- An array is a container object that holds a group of values of a single type.
- The type of the values in the array can be a primitive or an object type.
- The length of an array is established when the array is created.
- After creation, the length of an array is fixed.
- Each item in an array is called an element.
- Each element is accessed by a numerical index.
- The index of the first element is 0 (zero).

One-Dimensional Arrays

Example:

```
int ageOne = 27;
int ageTwo = 12;
int ageThree = 82;
int ageFour = 70;
int ageFive = 54;
int ageSix = 6;
int ageSeven = 1;
int ageEight = 30;
int ageNine = 34;
int ageTen = 42;
```


Consider a program where you store the ages of 10 people. You could create individual variables to hold each of the 10 values. You could do this using the code shown in the slide, but there are problems with this approach. What if you had to store 1,000 ages or 10,000 ages? As the number of values increases, your program becomes increasingly unmanageable. Or, what if you had to find the average age, or sort the ages into ascending order? You'd have to refer to each variable individually in your code.

As you will see, arrays in Java (and related constructs like Lists) give you a much more convenient way to work with sets of data. In this lesson, you learn about arrays. In the next lesson, you learn how to use loops to programmatically work through all the values in an array.

Creating One-Dimensional Arrays

Array of `int` types

Array of Shirt types

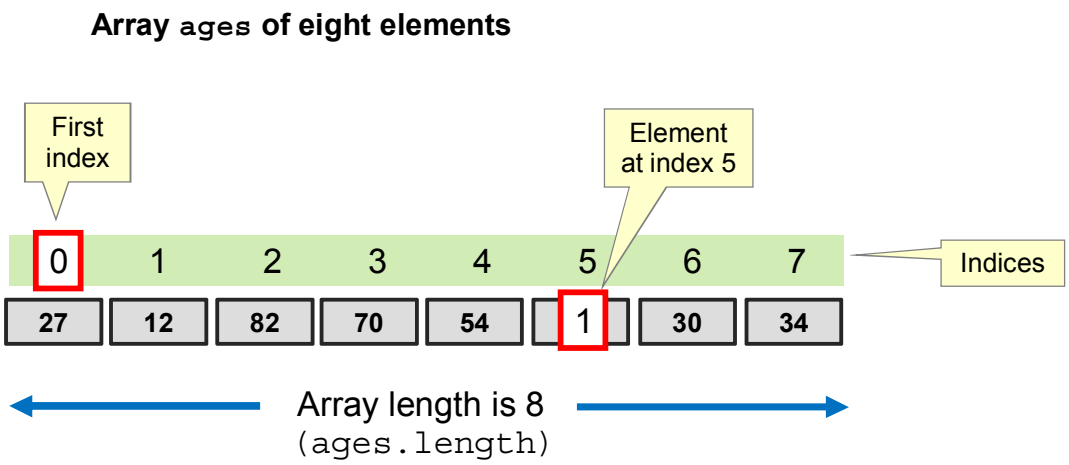
Array of String types

The Java programming language allows you to group multiple values of the same type (lists) using arrays. Arrays are useful when you have related pieces of data (such as the ages of several people), but you do not want to create separate variables to hold each piece of data.

You can create an array of primitive types, such as `int`, or an array of references to object types, such as Shirt or String. Each part of the array is an element. If you declare an array of 100 `int` types, there are 100 elements. You can access each specific element within the array by using its location or index in the array.

The diagram in the slide shows examples of arrays for `int` types, Shirt types, and String types.

Array Indices and Length



An array is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

Each item in an array is called an element, and each element is accessed by its numerical index. As shown in the diagram in the slide, numbering begins with 0. The eighth element, for example, would therefore be accessed at index 7.

The length of an array can be accessed using dot notation to access the `length` field. Assuming the array in the diagram is called `ages`, you can use:

```
int agesLength = ages.length;
```

This will assign a value of `8` to `int agesLength`.

Topics

- Overview of arrays
- Declaring, instantiating, and initializing arrays
- Accessing command-line arguments
- Working with two-dimensional arrays
- Working with ArrayList

Declaring a One-Dimensional Array

- Syntax:

```
type [] array_identifier;
```

- Declare arrays of types `char` and `int`:

```
char [] status;
int [] ages;
```

- Declare arrays of object references of types Shirt and String:

```
Shirt [] shirts;
String [] names;
```


Arrays are handled by an implicit Array object (which is not available in the Java API). Just like with any object, you must declare an object reference to the array, instantiate an Array object, and then initialize the Array object before you can use it.

The syntax used to declare a one-dimensional array is:

```
type [] array_identifier;
```

where:

- The `type` represents the primitive data type or object type for the values stored in the array
- The `[]` informs the compiler that you are declaring an array
- The `array_identifier` is the name that you are assigning to refer to the array

When you declare an array, the compiler and the Java Virtual Machine (JVM) have no idea how large the arrays will be because you have declared a reference variable that does not currently point to any objects.

Instantiating a One-Dimensional Array

- Syntax:

```
array_identifier = new type [length];
```

- Examples:

```
status = new char [20];
ages = new int [5];

names = new String [7];
shirts = new Shirt [3];
```


Before you can initialize an array, you must instantiate an Array object large enough to hold all of the values in the array. Instantiate an array by defining the number of elements in the array.

The syntax used to instantiate an Array object is:

```
array_identifier = new type [length];
```

where:

- The `array_identifier` is the name you are assigning to reference the array
- The `type` represents the primitive data type or object type for the values stored in the array
- The `length` represents the size (in number of elements) of the array

When you instantiate an Array object, every primitive element is initialized to the zero value for the type you specified. In the case of the `char` array called `status`, each value is initialized to `\u0000` (the null character of the Unicode character set). For the `int` array called `ages`, the initial value is the integer value `0`. For the `names` and `shirt` arrays, the object references are initialized to null.

Initializing a One-Dimensional Array

- Syntax:

```
array_identifier[index] = value;
```

- Set values in the `ages` array:

```
ages[0] = 19;
ages[1] = 42;
ages[2] = 92;
ages[3] = 33;
```

- Set references to Shirt objects in the `shirts` array:

```
shirts[0] = new Shirt();
shirts[1] = new Shirt();
shirts[2] = new Shirt();
```


You can fill the contents of an array after you have created it. The syntax for setting the values in an array is:

```
array_identifier[index] = value;
```

where:

- The `array_identifier` is the name you are assigning to the array
- The `index` represents the location in the array where the value will be placed

Use the `new` keyword to create the Shirt objects and to place the references to the Shirt objects into each position in the array.

Note: The index to the first element of an array is 0 and the index to the last element of the array is the length of the array minus 1. For example, the last element of a six-element array is index 5.

Declaring, Instantiating, and Initializing One-Dimensional Arrays

- Syntax:

```
type [] array_identifier = {comma-separated list of values
or expressions};
```

- Examples:

```
int [] ages = {19, 42, 92, 33, 46};
Shirt [] shirts = {new Shirt(), new Shirt(), new Shirt()};
```

- Not permitted (NetBeans will show an error):

```
int [] ages;
ages = {19, 42, 92, 33, 46};
```


If you know the values you want in your array at the time that you declare it, you can declare, instantiate, and set the values for an Array object in the same line of code. The syntax for this is:

```
type [] array_identifier =
                          {comma-separated_list_of_values_or_expressions};
```

where:

- The `type` represents the primitive data type or object type for the values to be stored
- The `[]` informs the compiler that you are declaring an array
- The `array_identifier` is the name you are assigning to the array
- The `{comma-separated_list_of_values_or_expressions}` represents a list of values that you want to store in the array

The examples in the slide show statements that combine the declaration, instantiation, and initialization. Notice how the `new` keyword is used to instantiate the Shirt object so that a reference to that object can be placed in the array.

The final example in the slide will return an error. You cannot declare and initialize an array in separate lines by using the comma-separated list technique.

Accessing a Value Within an Array

- Setting a value:

```
status[0] = '3';
names[1] = "Fred Smith";
ages[1] = 19;
prices[2] = 9.99F;
```

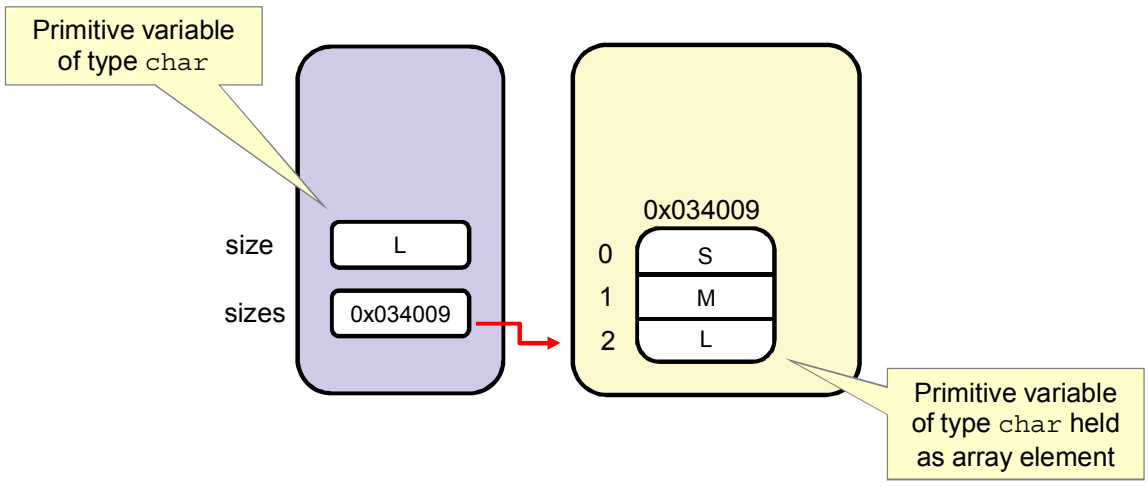
- Getting a value:

```
char s = status[0];
String name = names [1];
int age = ages[1];
double price = prices[2];
```


Each element of an array is accessed using its index. To access a value from the array, state the array name and the index number for the element (in brackets `[]`) on the right side of an assignment operator.

Storing Arrays in Memory

```
char size = 'L'
char[] sizes = {'S', 'M', 'L' };
```

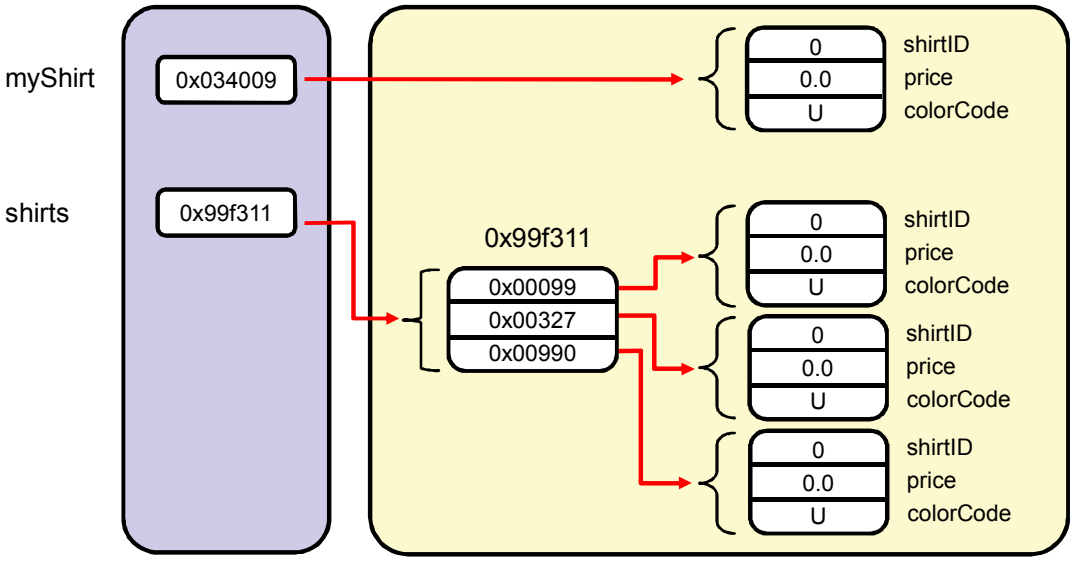


Arrays are objects referred to by an object reference variable. The diagram in the slide illustrates how a primitive array is stored in memory in comparison to how a primitive data type is stored in memory.

The value of the size variable (a `char` primitive) is `L`. The value of `sizes[]` is `0x334009`, and it points to an object of type `array` (of `char` types) with three values. The value of `sizes[0]` is `char S`, the value of `sizes[1]` is `char M`, and the value of `sizes[2]` is `char L`.

Storing Arrays of References in Memory

```
Shirt myShirt = new Shirt();
Shirt[] shirts = { new Shirt(), new Shirt(), new Shirt() };
```



The diagram in the slide illustrates how an object reference array is stored in memory. The value of the myShirt object reference is x034009, which is an address to an object of type Shirt with the values 0, 0.0, and U. The value of the shirts[ ] object reference is x99f311, which is an address to an object of type `array` (of Shirt object references) containing three object references:

- The value of the shirts[0] index is 0x00099, which is an object reference pointing to an object of type Shirt.
- The value of the shirts[1] index is 0x00327, which is an object reference pointing to another object of type Shirt.
- The value of the shirts[2] index is 0x00990, which is an object reference pointing to another object of type Shirt.

Quiz

The following code is the correct syntax for _____ an array:

```
array_identifier = new type [length];
```

a. Declaring
b. Setting array values
c. Instantiating
d. Declaring, instantiating, and setting array values

Answer: c

Quiz

Given the following array declaration, determine which of the three statements below it are true.

```
int [ ] autoMobile = new int [13];
```

a. `autoMobile[0]` is the reference to the first element in the array.
b. `autoMobile[13]` is the reference to the last element in the array.
c. There are 13 integers in the autoMobile array.

Answer: a, c

Topics

- Overview of arrays
- Declaring, instantiating, and initializing arrays
- Accessing command-line arguments
- Working with two-dimensional arrays
- Working with ArrayList

Using the `args` Array in the `main` Method

- Parameters can be typed on the command line.

```
> java ArgsTest Hello World!
args[0] is Hello
args[1] is World!
```

The second parameter goes into `args[1]` and so on.

The first parameter goes into `args[0]`.

- Code for retrieving the parameters.

```
public class ArgsTest {
   public static void main (String args[]) {
      System.out.println("args[0] is " + args[0]);
      System.out.println("args[1] is " + args[1]);
   }
}
```

ORACLE

When you pass strings to your program on the command line, the strings are put in the `args` array. To use these strings, you must extract them from the `args` array and, optionally, convert them to their proper type (because the `args` array is of type String).

The `ArgsTest` class shown in the slide extracts two String arguments passed on the command line and displays them.

To add parameters on the command line, you must leave one or more spaces after the class name (in this case, `ArgsTest`) and one or more spaces between each parameter added.

NetBeans does not allow you a way to run a Java class from the command line, but you can set command-line arguments as a property of the project your code is in. You use this technique in the practice for this lesson.

Converting String Arguments to Other Types

- Numbers can be typed as parameters.

```
> java ArgsTest 2 3
Total is: 23
Total is: 5
```

Concatenation, not addition!

- Conversion of String to `int`

These are Strings!

```
public class ArgsTest {
    public static void main (String args[]) {
        System.out.println("Total is: " + (args[0] + args[1]));

        int arg1 = Integer.parseInt(args[0]);
        int arg2 = Integer.parseInt(args[1]);
        System.out.println("Total is: " + (arg1 + arg2));
    }
}
```

Integer.parseInt() converts to int.

Note parentheses.

ORACLE

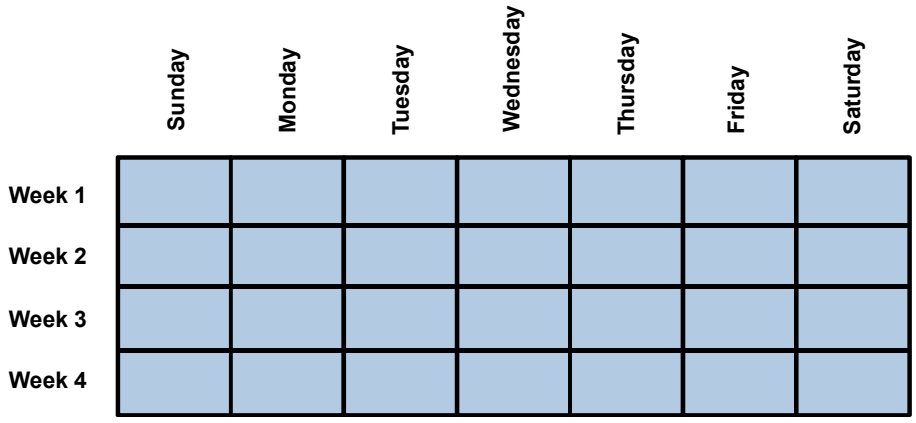
The `main` method treats everything you type as a literal string. If you want to use the string representation of a number in an expression, you must convert the string to its numerical equivalent. Every data type has an associated class containing static utility methods for converting strings to that data type (`Integer` class for `int`, `Byte` class for `byte`, `Long` class for `long`, and so on). For example, to convert the first argument passed to the `main` method to an `int` type, use `Integer.parseInt(args[])`.

Note that the parentheses around `arg1 + arg2` are required so that the `+` sign indicates addition rather than concatenation.

Topics

- Overview of arrays
- Declaring, instantiating, and initializing arrays
- Accessing command-line arguments
- Working with two-dimensional arrays
- Working with ArrayList

Describing Two-Dimensional Arrays



You also can store matrices of data by using multi-dimensional arrays (arrays of arrays, of arrays, and so on). A two-dimensional array (an array of arrays) is similar to a spreadsheet with multiple columns (each column represents one array or list of items) and multiple rows.

The diagram in the slide shows a two-dimensional array. Note that the descriptive names Week 1, Week 2, Monday, Tuesday, and so on would not be used to access the elements of the array. Instead, Week 1 would be index 0 and Week 4 would be index 3 along that dimension, while Sunday would be index 0 and Saturday would be index 6 along the other dimension.

Declaring a Two-Dimensional Array

- Syntax:

```
type [][] array_identifier;
```

- Example:

```
int [][] yearlySales;
```


Two-dimensional arrays require an additional set of square brackets. The process of creating and using two-dimensional arrays is otherwise the same as with one-dimensional arrays. The syntax for declaring a two-dimensional array is:

```
type [][] array_identifier;
```

where:

- The `type` represents the primitive data type or object type for the values stored in the array
- The `[][]` inform the compiler that you are declaring a two-dimensional array
- The `array_identifier` is the name you have assigned the array during declaration

The example shown declares an array of arrays for quarterly sales amounts over five years.

Instantiating a Two-Dimensional Array

- Syntax:

```
array_identifier = new type [number_of_arrays] [length];
```

- Example:

```
// Instantiates a 2D array: 5 arrays of 4 elements each
yearlySales = new int[5][4];
```

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1				
Year 2				
Year 3				
Year 4				
Year 5				

The syntax for instantiating a two-dimensional array is:

```
array_identifier = new type [number_of_arrays] [length];
```

where:

- The `array_identifier` is the name you have assigned the array during declaration
- The `number_of_arrays` is the number of arrays within the array
- The `length` is the length of each array within the array

The example shown in the slide instantiates an array of arrays for quarterly sales amounts over five years. The `yearlySales` array contains five elements of the type `int` array (five sub-arrays). Each sub-array is four elements in size and tracks the sales for one year over four quarters.

Initializing a Two-Dimensional Array

Example:

```
yearlySales[0][0] = 1000;
yearlySales[0][1] = 1500;
yearlySales[0][2] = 1800;
yearlySales[1][0] = 1000;
yearlySales[3][3] = 2000;
```

	Quarter 1	Quarter 2	Quarter 3	Quarter 4
Year 1	1000	1500	1800	
Year 2	1000			
Year 3				
Year 4				2000
Year 5				

ORACLE

When setting (or getting) values in a two-dimensional array, indicate the index number in the array by using a number to represent the row, followed by a number to represent the column. The example in the slide shows five assignments of values to elements of the `yearlySales` array.

Topics

- Overview of arrays
- Declaring, instantiating, and initializing arrays
- Accessing command-line arguments
- Working with two-dimensional arrays
- Working with ArrayList

The ArrayList Class

Arrays are not the only way to store lists of related data:

- ArrayList is one of a number of list classes.
- It has a set of useful methods for managing its elements:
  - `add()`, `get()`, `remove()`, `indexOf()` and many others
- You do not need to specify a size when you instantiate an ArrayList:
  - As you add more elements, the ArrayList will grow as necessary.
  - You can specify an initial capacity, but it is not mandatory to do so.
- It can store only objects, not primitives.

ORACLE

For lists that are very dynamic, it may be easier to work with a specialized List type object. This can free you from having to write code to:

- Keep track of the index of the last piece of data added
- Keep track of how full the array is and determine if it needs to be resized
- Increase the size of the array by creating a new one and copying the elements from the current one into it

Class Names and the Import Statement

- ArrayList is in the package `java.util`.
- To refer to ArrayList in your code, you can fully qualify
  `java.util.ArrayList myList;`
  Or,
- Add the import statement at the top of the class.

```
import java.util.ArrayList;
public class ArrayListExample {
   public static void main (String args[]) {
       ArrayList myList;
   }
}
```

ORACLE

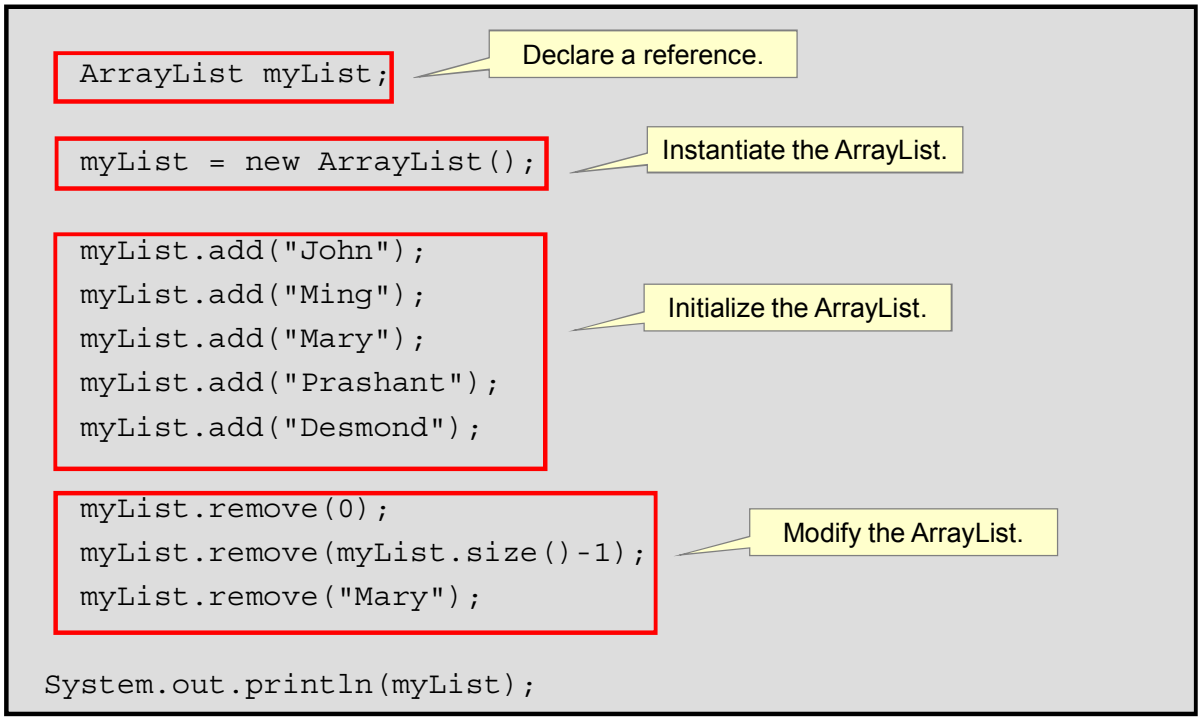
Classes in the Java programming language are grouped into packages depending on their functionality. For example, all classes related to the core Java programming language are in the `java.lang` package, which contains classes that are fundamental to the Java programming language, such as String, Math, and Integer. Classes in the `java.lang` package can be referred to in code by just their class name. They do not require full qualification or the use of an import statement.

All classes in other packages (for example, ArrayList) require that you fully qualify them in the code, or that you use an import statement so that they can be referred to directly in the code.

The import statement can be:

- For just the class in question
  `java.util.ArrayList;`
- For all classes in the package
  `java.util.*;`

Working with an ArrayList



Declaring an ArrayList is exactly the same as declaring any other reference type. Likewise, instantiating an ArrayList is the same as instantiating any other object (you can check the documentation for other possibilities for instantiating).

There are a number of methods to add data to the ArrayList. In the example in the slide, we use the simplest, `add()`, to add a string. Each call to `add` adds a new element to the end of the ArrayList.

Finally, a big advantage of ArrayList over an array is that there are many methods available for manipulating the data. The example here shows just one method, but it is very powerful.

- `remove(0)`: This removes the first element (in this case, `"John"`).
- `remove(myList.size() - 1)`: This removes the last element as `myList.size()` gives the number of elements of the array so the last one is the size minus 1 (this removes `"Desmond"`).
- `remove("Mary")`: This removes a specific element. In this case, we have the convenience of referring not to where the element is in the ArrayList, but rather to what it is.

You can pass an ArrayList to `System.out.println()` and the resulting output will be:

```
[Ming, Prashant]
```

Quiz

A two-dimensional array is similar to a _____:

a. Shopping list
b. List of chores
c. Matrix
d. Bar chart containing the dimensions for several boxes

Answer: c

Summary

- An array in Java is a data type that is composed of a set of other data types:
  - The data types can be objects or primitives.
  - Each data value is an element of the array.
- Arrays are created with a specific size (number of elements).
- Each element in an array can be accessed using its index:
  - The first index is 0 (zero).
- The data type of an array can be another array:
  - This creates a two-dimensional array.
- Another option is to use a specialized List class such as ArrayList.

Practice 8-1: Creating a Class with a One-Dimensional Array of Primitive Types

This practice covers creating an array to define employee vacation days at Duke's Choice.

In this practice, you create an array containing the number of vacation days that an employee at the Duke's Choice Company receives, based upon the number of years that the employee has worked for Duke's Choice.

Practice 8-2: Create and Work with an ArrayList

This practice covers the following topics:

- Populating and displaying an ArrayList
- Manipulating an ArrayList

In this practice, you create the NamesList class and the NamesListTest class to experiment with populating and manipulating ArrayLists.

Practice 8-3: Use Runtime Arguments and Parse the `args` Array

This practice covers the following topics:

- Creating a class that accepts a runtime argument
- Randomly generating a number in the class
- Comparing argument value with the randomly generated number

In this practice, you write a guessing game that accepts an argument and displays an associated message. You create a class that accepts a runtime argument between 1 and 5, inclusive. You also randomly generate a number between 1 and 5 in the class and compare the value of the argument with the randomly generated number.

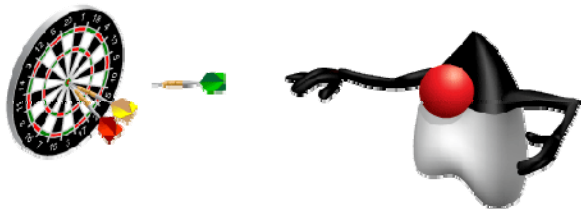
9

Using Loop Constructs

Objectives

After completing this lesson, you should be able to:

- Create a `while` loop
- Nest a `while` loop
- Develop and nest a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs
- Use an ArrayList in a `for` loop



ORACLE

Topics

- Create a `while` loop
- Develop a `for` loop
- Nest a `for` loop and a `while` loop
- Use an array in a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs

Loops

Loops are frequently used in programs to repeat blocks of statements until an expression is false.

- There are three main types of loops:
  - `while` loop
    - Repeats while an expression is true
  - `do/while` loop
    - Executes once, and then continues to repeat while true
  - `for` loop
    - Repeats a set number of times

Repeating Behavior


```
while (!areWeThereYet) {

   read book;
   argue with sibling;
   ask, "Are we there yet?";

}

Woohoo!;
Get out of car;
```


In computer programming it is a common task to need to repeat a number of statements. Typically, the code will continue to repeat the statements until something changes. Then, the code breaks out of the loop and continues with the next statement.

Creating `while` Loops

Syntax:

```
while (boolean_expression) {

   code_block;

}  // end of while construct

// program continues here
```

if Boolean expression is true, this code block executes.

if Boolean expression is false, program continues here.

`while` Loop in Elevator

```
public void setFloor() {
// Normally you would pass the desiredFloor as an argument to the
// setFloor method. However, because you have not learned how to
// do this yet, desiredFloor is set to a specific number (5)
// below.

   int desiredFloor = 5;
   while ( currentFloor != desiredFloor ){
      if (currentFloor < desiredFloor) {
         goUp();
      }
      else {
         goDown();
      }
   }
}
```

If Boolean expression returns true, execute `while` loop.

The code in the slide shows a very simple `while` loop in the `Elevator` class. Remember that this particular elevator accepts commands only for going up or down one floor at a time. So in order to move a number of floors, the `goUp()` or `goDown()` methods need to be called a number of times.

Notice how the Boolean expression is written. The expression returns true if `currentFloor` is not equal to `desiredFloor`. So, when these two variables are equal, this expression will return false (because the elevator is now at the desired floor), and the `while` loop will not be executed.

Types of Variables

```
public class Elevator {
   public boolean doorOpen=false;
   public int currentFloor = 1;
   public final int TOP_FLOOR = 5;
   public final int BOTTOM_FLOOR = 1;

   ... < lines of code omitted > ...

  public void setFloor() {
    int desiredFloor = 5;
    while (  currentFloor != desiredFloor  ){
       if (currentFloor < desiredFloor) {
          goUp();
       } else {
          goDown();
       }
    } // end of while loop
  } // end of method
} // end of class
```

Instance variables (fields)

local variable

Scope of `desiredFloor`

This `setFloor` method uses two different types of variables. The variable `currentFloor` is an instance variable, usually called a field. It is a member of the class. In an earlier lesson, you saw how fields of an object could be accessed by using the dot notation. Fields are declared outside of method code, usually just after the class declaration.

The variable `desiredFloor` is a local variable, declared within the `setFloor` method, and only accessible within that method. Another way to say this is that its scope is the `setFloor` method. As you will see later, local variables can also be declared within loops or `if` statements. Irrespective of whether declared within a method, loop, or `if` statement, the scope of a local variable is always the block within which it is declared.

`while` Loop: Example 1

Example:

```
float square = 4;    // number to find sq root of
float squareRoot = square;    // first guess
while (squareRoot * squareRoot - square > 0.001) { // How accurate?
    squareRoot = (squareRoot + square/squareRoot)/2;
    System.out.println("Next try will be " + squareRoot);
}
System.out.println("Square root of " + square + " is " + squareRoot);
```

Result:

```
Next try will be 2.5
Next try will be 2.05
Next try will be 2.0006099
Next try will be 2.0
The square root of 4.0 is 2.0
```


The example shows some code for generating the square root of number. The Boolean expression squares the current value of the square root and checks to see if it is close to the number we are trying to find the square root of. If it's close enough (the expression returns true), the program execution skips the statements in the `while` block and continues with the `System.out.println()` statement that outputs the square root. If the value is not yet close enough, the code within the block runs and does two things:

1. Adjusts the value of `squareRoot`, so that it will be closer the next time it's checked
2. Prints out the current "guessed" value of `squareRoot`

`while` Loop: Example 2

Example:

```
int initialSum = 500;
int interest = 7;          // per cent
int years = 0;
int currentSum = initialSum * 100; // Convert to pennies
   while ( currentSum <= 100000 ) {
      currentSum += currentSum * interest/100;
      years++;
      System.out.println("Year " + years + ": " + currentSum/100);
   }
```

Check if money has doubled yet.

If not doubled, add another year's interest.

Result:

```
 ... < some results not shown > ...
Year 9: 919
Year 10: 983
Year 11: 1052
```

`while` loop iterates 11 times before the Boolean test evaluates to true.

Here's an example that shows how long it would take to double your money at a particular interest rate. The `while` loop's Boolean expression checks to see if your money (converted to pennies) has doubled. If it hasn't, the block of the loop adds the interest of another year on to the current total and the loop repeats the Boolean expression check.

Note: Converting to pennies is done to simplify the example so that the `int` type can be used.

`while` Loop with Counter

Example:

```
System.out.println("    /*");
int counter = 0;
while ( counter < 4 ) {
   System.out.println("     *");
   counter ++;
}
System.out.println("     */");
```

Declare and initialize a counter variable.

Check to see if counter has exceeded 4.

Print out an asterisk and increment the counter.

Output:

```
/*
 *
 *
 *
 *
 */
```

ORACLE

Loops are often used to repeat a set of commands a specific number of times. You can easily do this by declaring and initializing a counter (usually of type `int`), incrementing that variable inside the loop, and checking if the counter has reached a specific value in the `while` Boolean expression.

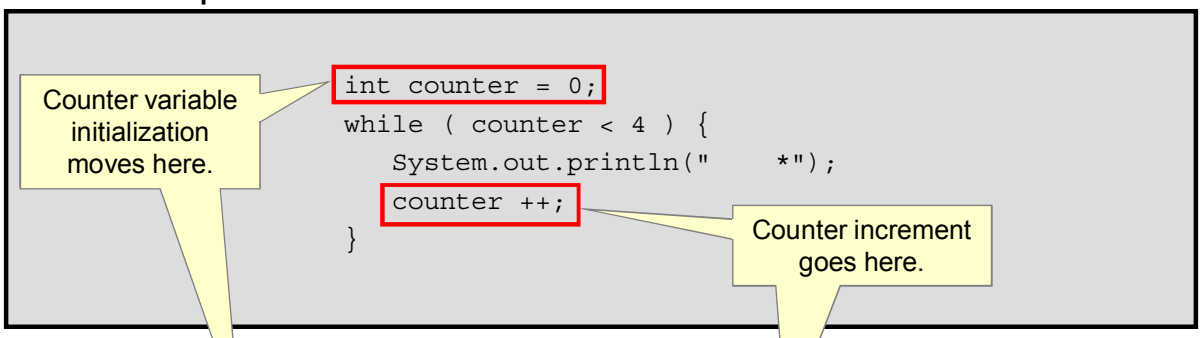
However, although this works, Java has a special counter loop (a `for` loop) that is covered in the following slides.

Topics

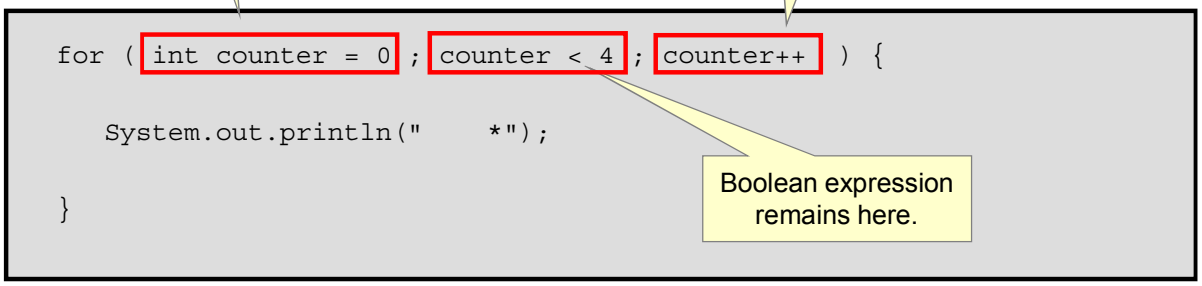
- Create a `while` loop
- Develop a `for` loop
- Nest a `for` loop and a `while` loop
- Use an array in a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs

for Loop

while loop:



for loop:



ORACLE

In the `for` loop, the three expressions needed for a loop that runs a set number of times are all moved into the parentheses after the `for` keyword. This makes the `for` loop more compact and readable.

Developing a `for` Loop

Syntax:

```
for (initialize[,initialize]; boolean_expression; update[,update]) {

    code_block;
}
```

Example:

```
for (String i = "|", t = "------";
     i.length() < 7 ;
     i += "|", t = t.substring(1) ) {

         System.out.println(i + t);

}
```

The three parts of the `for` loop

Notice that `for` loops are very versatile; you can initialize more than one variable in the first part, and modify more than one variable in the third part of the `for` statement. Also, the type need not be an `int`.

The code in the slide declares two Strings and, as it loops, appends to one String while removing from the other String. These changes are in the third part of the `for` statement. This part is for updates and, although often used for incrementing the String, can be used for any kind of update as shown here.

The output of the loop is:

```
|------
||-----
|||----
||||---
|||||--
||||||-
```

Topics

- Create a `while` loop
- Develop a `for` loop
- Nest a `for` loop and a `while` loop
- Use an array in a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs

Nested `for` Loop

Code:

```
int height = 4;
int width = 10;

for (int rowCount = 0; rowCount < height; rowCount++ ) {

   for (int colCount = 0; colCount < width; colCount++ ) {
       System.out.print("@");
   }
   System.out.println();
}
```


The code in the slide shows a simple nested loop to output a block of `@` symbols with height and width given in the initial local variables. Notice how the outer code prints a new line to start a new row, while the inner loop uses the `print()` method of `System.out` to print an `@` symbol for every column.

Nested `while` Loop

Code:

```
String name = "Lenny";
String guess = "";
int numTries = 0;

while (!guess.equals(name.toLowerCase())) {
    guess = "";
    while (guess.length() < name.length()) {
       char asciiChar = (char)(Math.random() * 26 + 97);
       guess = guess + asciiChar;
    }
    numTries++;
}
System.out.println(name + " found after " + numTries + " tries!");
```


Here's a nested `while` loop that's a little more complex than the previous `for` example. The nested loop tries to guess a name by building a String of the same length completely at random.

Looking at the inner loop first, the code initializes the `char asciiChar` to a lowercase letter randomly. These `chars` are then added to the String guess, until that String is as long as the String that it's being matched against. Notice the convenience of the concatenation operator here, allowing concatenation of a String and a `char`.

The outer loop tests to see if the guess is the same as a lowercase version of the original name. If it isn't, guess is reset to an empty String and the inner loop runs again, usually millions of times for a five-letter name. (Note that names longer than five letters will take a very long time!)

Topics

- Create a `while` loop
- Develop a `for` loop
- Nest a `for` loop and a `while` loop
- Use an array in a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs

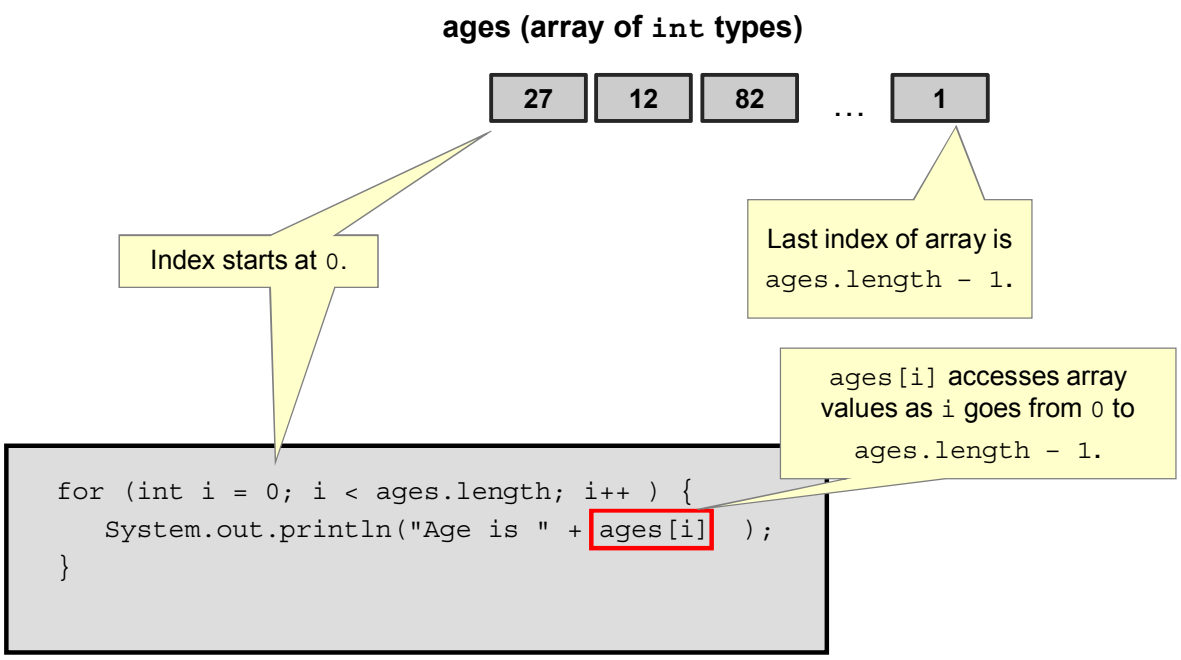
Loops and Arrays

One of the most common uses of loops is when working with sets of data.

All types of loops are useful:

- `while` loops if checking for a particular value
- `for` loops to go through the entire array
- Enhanced `for` loops

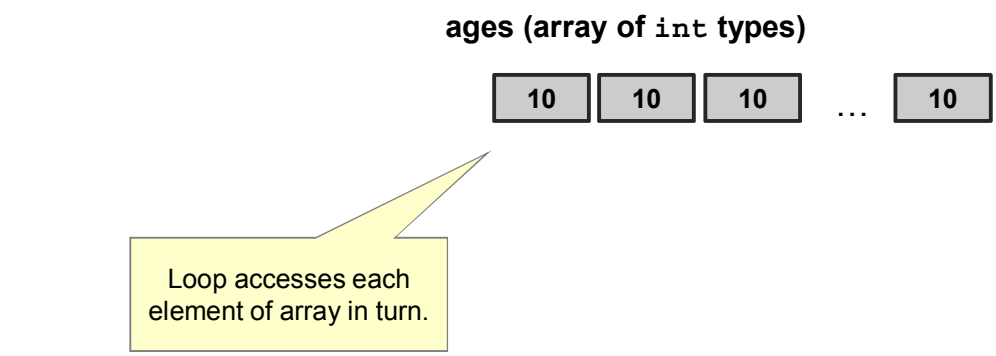
`for` Loop with Arrays



Output will be:

```
Age is 27
Age is 12
Age is 82
  …
Age is 1
```

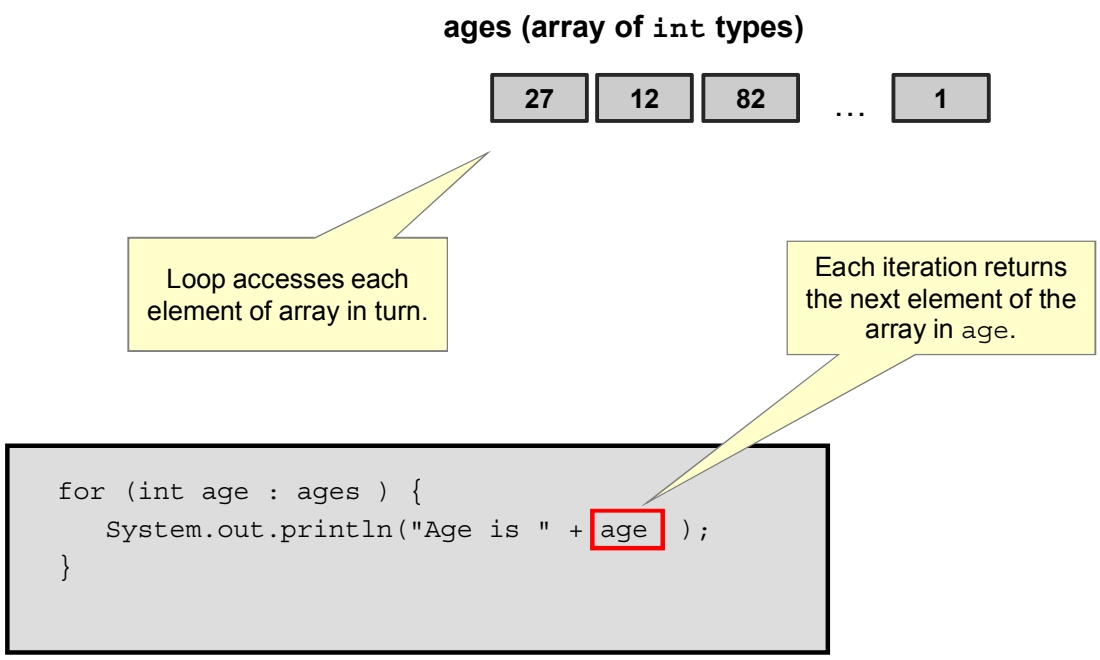
Setting Values in an Array



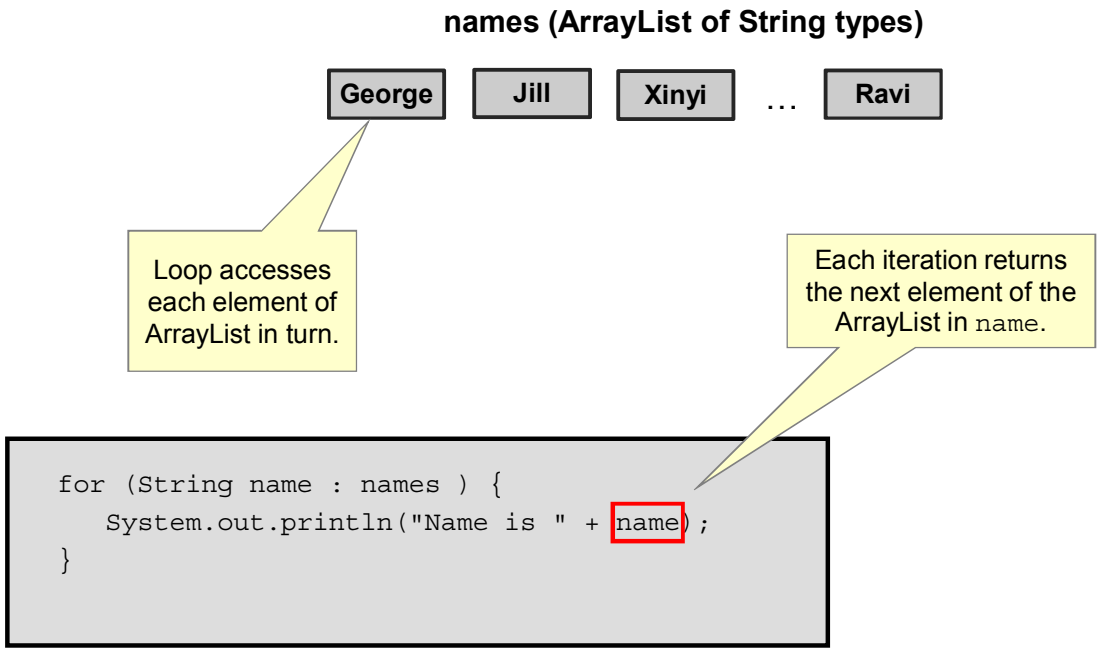

```
for (int i = 0; int < ages.length; i++ ) {
   ages[i] = 10;
}
```

Each element in the array is set to `10`.

Enhanced `for` Loop with Arrays



Enhanced `for` Loop with ArrayLists



ArrayLists can be iterated through in exactly the same way as arrays.

Using `break` with Loops

`break` example:

```
int passmark = 12;
boolean passed = false;
int[] score = { 4, 6, 2, 8, 12, 34, 9 };
for (int unitScore : score ) {
    if ( unitScore > passmark ) {
       passed = true;
       break;
    }
}
System.out.println("One or more units passed? " + passed);
```

There is no need to go through the loop again, so use break.

Output:

```
One or more units passed? true
```


There are two useful keywords that can be used when working with loops: `break` and `continue`. `break` allows you to jump out of a loop, while `continue` sends you back to the start of the loop.

The example in the slide shows the use of `break`. Assuming that the code is to find out if any of the scores in the array are above the `passmark`, we can set `passed` to `true` and jump out of the loop as soon as the first such score is found.

Using `continue` with Loops

`continue` example:

```
int passMark = 15;
int passesReqd = 3;
int[] score = { 4, 6, 2, 8, 12, 34, 9 };
for (int unitScore : score ) {
    if (score[i] < passMark) {
        continue;
    }
    passesReqd--;
    // Other processing
 }
 System.out.println("Units still reqd " + Math.max(0,passesReqd));
```

If unit failed, go on to check next unit.

This example shows the use of `continue` on a similar example. In this case, assume that we want to know if a certain number of passes has been achieved. So the approach is to check first to see if the unit's score is not enough. If this is the case, the `continue` command goes to the start of the loop again. If the score is sufficient, then the number of `passesReqd` is decremented and possibly further processing takes place.

This example and the previous one are intended only to show what the functions of `break` and `continue` are, and not to show particular programming techniques. Both have a similar function; they ensure that parts of the loop are not processed unnecessarily. Sometimes this can also be achieved by the design of `if` blocks, but in complex algorithms it is useful to have these two options.

Topics

- Create a `while` loop
- Develop a `for` loop
- Nest a `for` loop and a `while` loop
- Use an array in a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs

Coding a `do/while` Loop

Syntax:

```
do {

    code_block;
}
while (boolean_expression); // Semicolon is mandatory.
```


The `do/while` loop is a one-to-many iterative loop: The condition is at the bottom of the loop and is processed *after the body.* The *body of the loop* is therefore processed at least once. If you want the statement or statements in the body to be processed at least once, use a `do/while` loop instead of a `while` or `for` loop. The syntax for the `do/while` loop is shown in the slide.

Coding a `do/while` Loop

```
setFloor() {
    // Normally you would pass the desiredFloor as an argument to the
   // setFloor method. However, because you have not learned how to
   // do this yet, desiredFloor is set to a specific number (5)
   // below.
     int desiredFloor = 5;

   do {
     if (currentFloor < desiredFloor) {
       goUp();
     }
     else if (currentFloor > desiredFloor) {
       goDown();
     }
   }
    while (currentFloor != desiredFloor);
 }
```

ORACLE

The `setFloor` method of the `Elevator` class uses a `do/while` loop to determine if the elevator is at the chosen floor. If the value of the `currentFloor` variable is not equal to the value of the `desiredFloor` variable, then the elevator continues moving either up or down.

Topics

- Create a `while` loop
- Develop a `for` loop
- Nest a `for` loop and a `while` loop
- Use an array in a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs

Comparing Loop Constructs

- Use the `while` loop to iterate indefinitely through statements and to perform the statements zero or more times.
- Use the `do/while` loop to iterate indefinitely through statements and to perform the statements *one* or more times.
- Use the `for` loop to step through statements a predefined number of times.

Quiz

Fill in the blank of the following sentence with one of the options given below.

_____ enable you to check and recheck a decision to execute and re-execute a block of code.

a. Classes
b. Objects
c. Loops
d. Methods

Answer: c

Quiz

Which of the following types of loops is a one/many iterative loop?

a. The `while` loop
b. The nested `while` loop
c. The `do/while` loop
d. The `for` loop

Answer: c

Summary

In this lesson, you should have learned how to:

- Create a `while` loop
- Nest a `while` loop
- Develop and nest a `for` loop
- Code and nest a `do/while` loop
- Compare loop constructs
- Use an ArrayList in a `for` loop

Practice 9.1 Overview: Writing a Class That Uses a `for` Loop

In this practice, you create the `Counter` class that uses a simple `for` loop to print a sequence of numbers.

Practice 9.2 Overview: Writing a Class That Uses a `while` Loop

In this practice, you write a class named `Sequence` that displays a sequence starting with the numbers 0 and 1. Successive numbers in the sequence are the sum of the previous two numbers (for example, 0 1 1 2 3 5 8 13 21…). This sequence is also called the Fibonacci series.

Practice 9.3 Overview: Using `for` Loops to Process an ArrayList

In this practice, you create two new methods in two different classes. This practice contains two sections:

- Using a `for` loop with the `VacationScaleTwo` class
- Using an enhanced `for` loop with the `NamesListTwo` class

Note: Practice 9.3 is an optional Challenge practice.

In this practice, you create two new methods in two different classes. One of the methods uses a traditional `for` loop to display the values in an ArrayList. The other method uses an *enhanced* `for` loop to display the values in the ArrayList.

Practice 9.4 Overview: Writing a Class That Uses a Nested `for` Loop to Process a Two-Dimensional Array

In this practice, you create and process a two-dimensional array using a nested `for` loop (one loop within another loop).

In this practice, you create and process a two-dimensional array using a nested `for` loop (one loop within another loop). This practice is based on the scenario of a classroom that has 12 desks arranged in a rectangular grid comprised of three rows and four columns. Students are allocated a desk at the position found vacant first, by traversing each row.

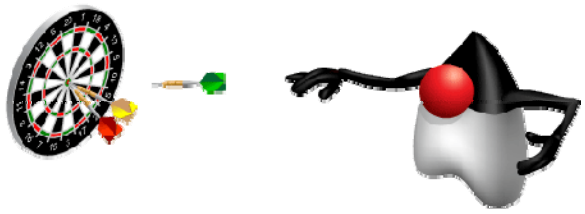
10

Working with Methods and Method Overloading

Objectives

After completing this lesson, you should be able to:

- Declare methods with arguments and return values
- Declare static methods and variables
- Create an overloaded method



ORACLE

Topics

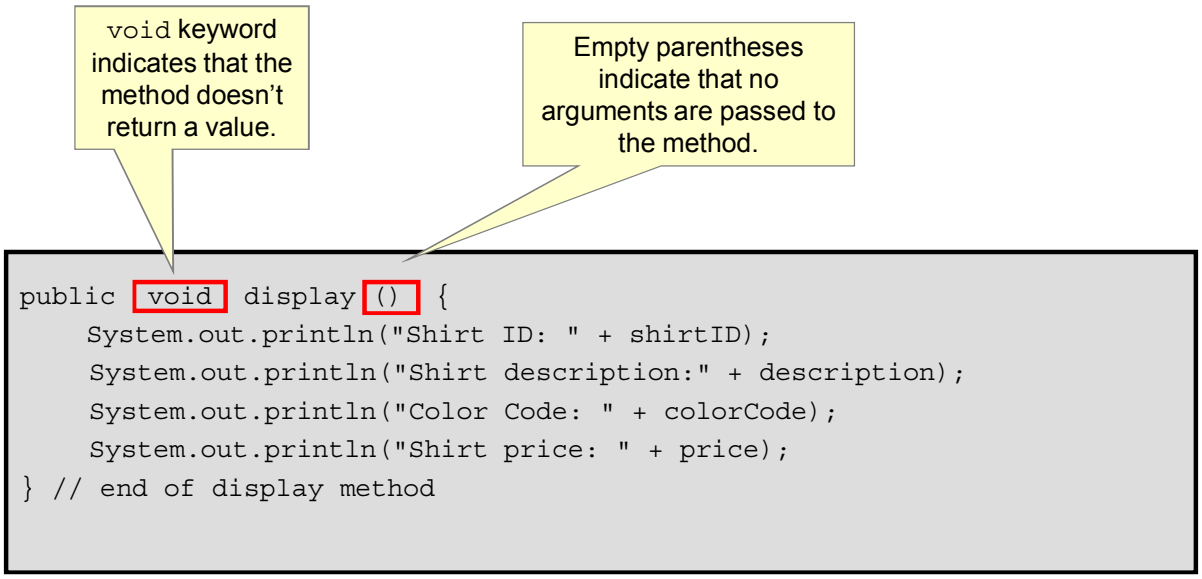
- Creating and invoking methods
- Static methods and variables
- Method overloading

Creating and Invoking Methods

Syntax:

```
[modifiers] return_type method_identifier ([arguments]) {
  method_code_block
}
```

Basic Form of a Method



ORACLE

This is an example of a simple method that doesn't receive any arguments or return a value.

Invoking a Method in a Different Class

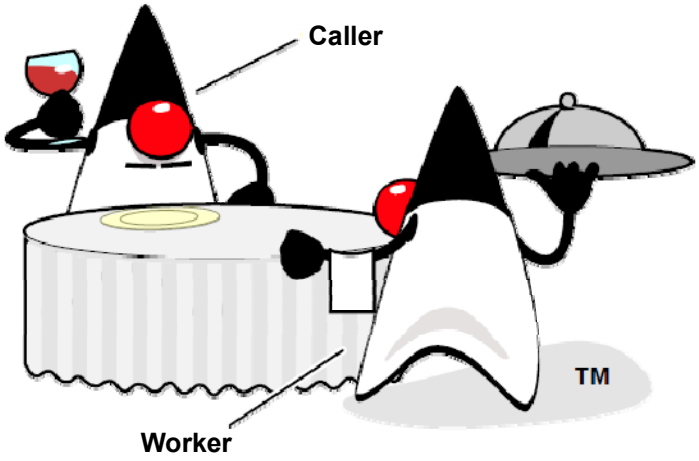
```
public class ShirtTest {
   public static void main (String args[]) {
      Shirt myShirt;
      myShirt = new Shirt();
      myShirt.display();
   }
}
```

Output:

```
Item ID: 0
Item description:-description required-
Color Code: U
Item price: 0.0
```


`display()` is being called, but because the Shirt object has not had any of its fields set, the default values for those fields are being displayed.

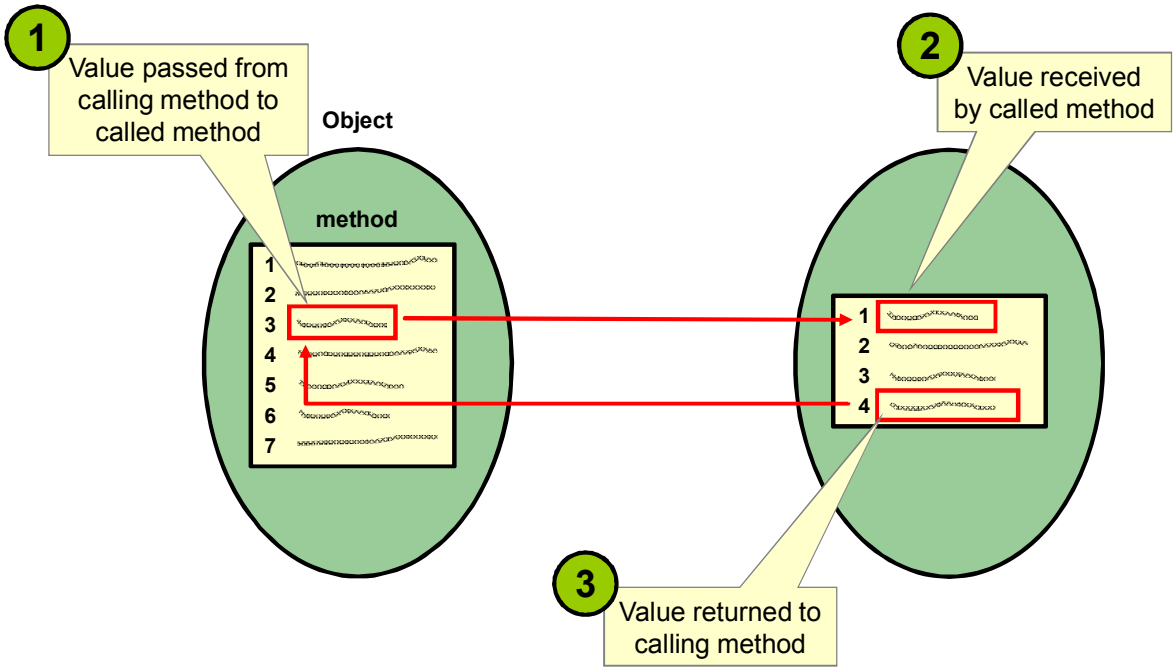
Calling and Worker Methods



In the previous example, the `ShirtTest` class calls the `display()` method from within another method (the `main` method). Therefore, the main method is referred to as the calling method because it is invoking or “calling” another method to do some work. Conversely, the `display` method is referred to as the worker method because it does some work for the `main` method.

When a calling method calls a worker method, the calling method stops execution until the worker method is done. After the worker method has completed, program flow returns to the point after the method invocation in the calling method.

Passing Arguments and Returning Values



ORACLE

Creating a Method with a Parameter

Caller:

```
Elevator theElevator = new Elevator();

theElevator.setFloor( 4 ); // Send elevator to the fourth floor
```

A call to the `setFloor()` method, passing the value 5, of type `int`

Worker:

```
public void setFloor( int desiredFloor ) {
   while (currentFloor != desiredFloor){
   if (currentFloor < desiredFloor){
      goUp();
   }
   else {
      goDown();
   }
}
```

The `setFloor()` method receives an argument of type `int`, naming it `desiredFloor`.

Here's the `setFloor()` method (introduced in the lesson titled "Using Loop Constructs"). The method receives a value of `int` type and gives it the name `desiredFloor`. `desiredFloor` is now a local variable whose scope is the method.

It is called (in this case, from a calling method in another class) by using the dot notation and including the argument.

Note: A variable defined in the method declaration is called a method parameter, whereas a value passed into the method call is called an argument.

Creating a Method with a Return Value

Caller:

```
... < lines of code omitted > ...

boolean isOpen = theElevator.checkDoorStatus() // Is door open?
```

The local variable `isOpen` indicates if elevator door is open.

Worker:

```
public class Elevator {
   public boolean doorOpen=false;
   public int currentFloor = 1;

   ... < lines of code omitted > ...

   public boolean checkDoorStatus() {

      return doorOpen ;
   }
}
```

Elevator has `doorOpen` field to indicate state of elevator door.

The type returned by the method is defined before the method name.

The return statement returns the value in `doorOpen`.

The example in the slide shows the `checkDoorStatus()` method being called by the caller method. Note how the `checkDoorStatus()` defines that it will return a Boolean. Any single type can be defined here, or the keyword `void` is used if the method does not return a value.

The value is returned to the calling statement by the return statement. Note that because the method has been declared with a return type of `boolean`, NetBeans will indicate an error if there is no return or if the return is of an incorrect type.

Invoking a Method in the Same Class

```
public class Elevator {

public boolean doorOpen=false;
public int currentFloor = 1;

public final int TOP_FLOOR = 5;
public final int BOTTOM_FLOOR = 1;

public void openDoor() {

    // Check if door already open
    if ( !checkDoorStatus() ) {

        // door opening code
    }
}
```

Evaluates to true if door is closed

Calling a method in the same class is very straightforward. You can simply use the method name without a reference and dot notation. This is the same as when accessing a field; you can simply use the field name.

However, if you have local variables with similar names and you want to make it obvious that your code is accessing a field or method of the current object, you can use the `this` keyword with dot notation. `this` is a reference to the current object.

Example:

```
this.checkDoorStatus()
```

Advantages of Method Use

Methods:

- Make programs more readable and easier to maintain
- Make development and maintenance quicker
- Are central to reusable software
- Allow separate objects to communicate and to distribute the work performed by the program

Quiz

Which of the following statements are true? (Choose all that apply.)

a. A class can contain only one single method declaration.
b. A method must always specify a return type.
c. The same method can be both a worker method and a calling method.
d. Arguments need not be listed in the same order in the method invocation as in the method signature.

Answer: b, c

Invoking Methods: Summary

- There is no limit to the number of method calls that a calling method can make.
- The calling method and the worker method can be in the same class or in different classes.
- The way you invoke the worker method is different depending on whether it is in the same class or in a different class from the calling method.
- You can invoke methods in any order.
  - Methods do not need to be completed in the order in which they are listed in the class where they are declared (the class containing the worker methods).

Topics

- Creating and invoking methods
- Static methods and variables
- Method overloading

Math Utilities

```
String name = "Lenny";
String guess = "";
int numTries = 0;

while (!guess.equals(name.toLowerCase())) {
    guess = "";
    while (guess.length() < name.length()) {
       char asciiChar = (char)(Math.random() * 26 + 97);
       guess = guess + asciiChar;
    }
    numTries++;
}
System.out.println(name + " found after " + numTries + " tries!");
```

Creates a random letter

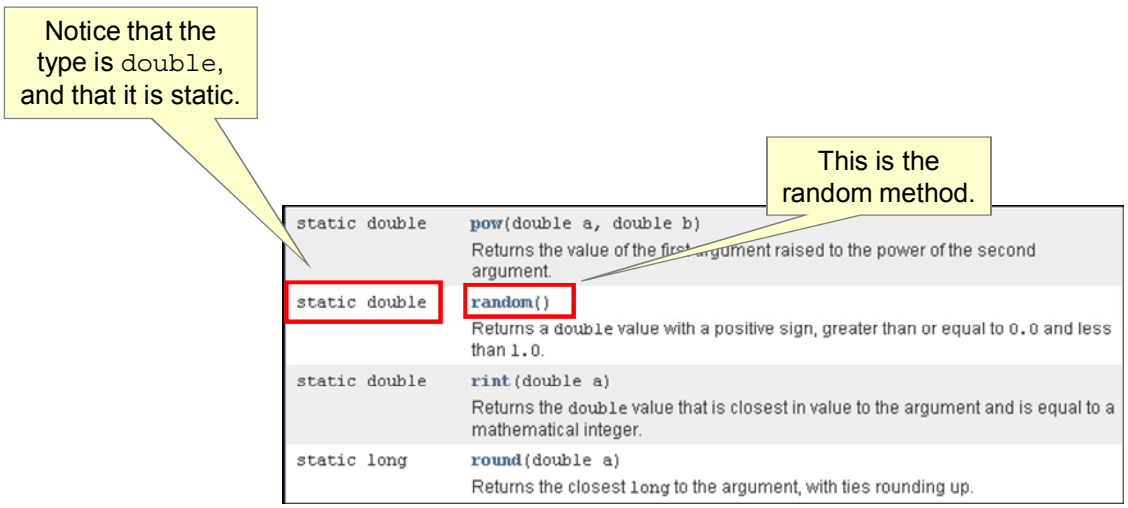
The code in the slide was used in the lesson on loops, but one part of it—the part where a random letter is generated—was not explained.

ASCII character values encode lowercase a to z from 97 to 122. By generating a number in that range and putting it into a `char`, you can use the concatenation operator to build a String as shown here.

Note: Java actually uses Unicode, not ASCII, but the first 128 characters in Unicode and ASCII are the same.

In the next slide, you'll look a little closer at the `Math.random()` method and what kind of method it is.

Static Methods in `Math`



ORACLE

The screenshot shows a small selection of methods from the `Math` class. The method we are focusing on here is `random()`. Its description is that it returns a double between 0 and 1. So to generate a double between 0 and 10, simply multiply by 10.

```
Math.random * 10
```

Or to generate a double between 1 and 10, multiply by 9 and add 1.

Of course, often you will want an integer rather than a double, so then all you need to do is cast to `int` or, in the case of the example on the previous page, to `char`.

Notice that the method is static, as indeed are all the methods in `Math`. That means that `Math` does not need to be instantiated to call any of its methods (in fact, `Math` cannot be instantiated).

You can call the static methods of a class with the following syntax:

```
<classname>.<method_name>
```

Creating `static` Methods and Variables

Methods and non-local variables can be static.

- They belong to the class and not the object.
- They are declared using the `static` keyword:

```
static Properties getProperties()
```

- Invoking `static` methods:

```
Classname.method();
```

- Accessing `static` variables in another class:

```
Classname.attribute_name;
```

- Accessing `static` variables in the same class:

```
attribute_name;
```

ORACLE

So far you have learned how to access methods and variables by creating an object of the class that the method or variable belongs to, and invoking the method or accessing the variable (if it is a public variable). Methods and variables that are unique to an instance are called instance methods and instance variables.

You have also been using methods that do not require object instantiation, such as the `main` method. These are called *class methods* or *static methods*; you can invoke them without creating an object first.

Similarly, the Java programming language allows you to create static variables or class variables, which you can use without creating an object.

Creating `static` Methods and Variables

```
public static char convertShirtSize(int numericalSize) {
    if (numericalSize < 10) {
      return 'S';
    }
    else if (numericalSize < 14) {
      return 'M';
    }

    else if (numericalSize < 18) {
      return 'L';
    }

    else {
      return 'X';
  }
}
```


The following is an example of a method that could be added to the `Shirt` class to convert numerical shirt sizes to sizes such as small, medium, or large. This method is a static method because:

- It does not directly use any attributes of the `Shirt` class
- You might want to invoke the method even if you do not have a Shirt object

The `convertShirtSize` method accepts a numerical size, determines the corresponding character size (S, M, L, or X), and returns the character size.

For example, to access the static method `convertShirtSize()` of `Shirt` class:

```
char size = Shirt.convertShirtSize(16);
```

`static` Variables

- Declaring `static` variables:

```
static double salesTAX = 8.25;
```

- Accessing `static` variables:

```
Classname.variable;
```

- Example:

```
double myPI;
myPI = Math.PI;
```


You can also use the `static` keyword to declare a class variable. This means that there can be only one copy of the variable in memory associated with a class, not a copy for each object instance.

In the example in the slide, `salesTAX` is a static variable. You can access it from any method in any class by using the class name of its class. Let's assume it is in a class called `TaxUtilities`. Then you could access it by using the code:

```
TaxUtilities.salesTAX
```

Or, if `TaxUtilities` has methods, those methods (static or instance) can access the variable by name without the class name:

```
salesTAX
```

Note that variables can have both the static and final modifier to indicate that there is only one copy of the variable and that the contents of the variable cannot be changed. The `PI` variable in the `Math` class is a static final variable.

Static Methods and Variables in the Java API

Examples

- Some functionality of the `Math` class:
  - Exponential
  - Logarithmic
  - Trigonometric
  - Random
  - Access to common mathematical constants such as the value pi (`Math.PI`)
- Some functionality of the `System` class:
  - Retrieving environment variables
  - Access to the standard input and output streams
  - Exiting the current program (`System.exit()`)

ORACLE

Certain Java class libraries, such as the `System` and the `Math` class, contain only static methods and variables. The `System` class contains utility methods for handling operating system–specific tasks (they do not operate on an object instance). For example, the `getProperties` method of the `System` class gets information about the computer that you are using.

The `Math` class contains utility methods for math operations.

Static Methods and Variables in the Java API

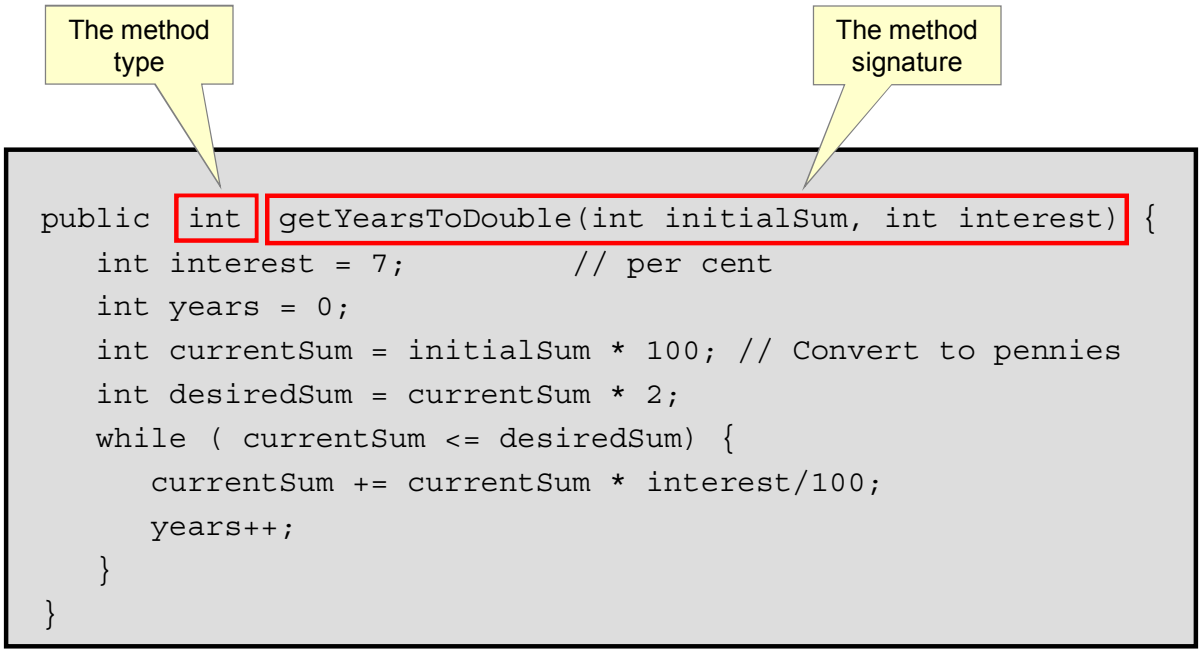
When to declare a `static` method or variable:

- Performing the operation on an individual object or associating the variable with a specific object type is not important.
- Accessing the variable or method before instantiating an object is important.
- The method or variable does not logically belong to an object, but possibly belongs to a utility class, such as the `Math` class, included in the Java API.

Topics

- Creating and invoking methods
- Static methods and variables
- Method overloading

Method Signature



The example shows some code from the lesson on loops, rewritten as a method that has two parameters (the initial sum of money and the interest rate) and returns the number of years required to double that initial sum.

The callout shows the part of the method declaration that is called the method signature.

The method signature of a method is the unique combination of the method name and the number, types, and order of its parameters. The method signature does not include the return type.

Method Overloading

Overloaded methods:

- Have the same name
- Have different signatures
  - Different number and/or different type and/or different order of parameters
- May have different functionality or similar functionality
- Widely used in the foundation classes

In the Java programming language, there can be several methods in a class that have the same name but different arguments (so the method signature is different). This concept is called method overloading. Just as you can distinguish between two students named Jim in the same class by calling them “Jim in the green shirt” and “Jim with the beeper,” you can distinguish between two methods by their name and arguments.

Using Method Overloading

```
public final class Calculator {

  public static int sum(int numberOne, int numberTwo){
     System.out.println("Method One");
     return numberOne + numberTwo;
  }

  public static float sum(float numberOne, float numberTwo) {
     System.out.println("Method Two");
     return numberOne + numberTwo;
  }
  public static float sum(int numberOne, float numberTwo) {
    System.out.println("Method Three");
    return numberOne + numberTwo;
  }
```


The example shows three methods to add two numbers, such as two `int` types or two `float` types. With method overloading, you can create several methods with the same name and different signatures.

The first `sum` method accepts two `int` arguments and returns an `int` value. The second `sum` method accepts two `float` arguments and returns a `float` value. The third `sum` method accepts an `int` and a `float` as arguments and returns a `float`.

To invoke any of the `sum` methods, the compiler compares the method signature in your method invocation against the method signatures in a class.

Using Method Overloading

```
public class CalculatorTest {

  public static void main(String [] args) {

    int totalOne = Calculator.sum(2,3);
    System.out.println("The total is " + totalOne);

    float totalTwo = Calculator.sum(15.99F, 12.85F);
    System.out.println(totalTwo);

    float totalThree = Calculator.sum(2, 12.85F);
    System.out.println(totalThree);
  }
}
```


The code example in the slide has a `main` method that invokes each of the previous `sum` methods of the Calculator class.

Method Overloading and the Java API

Method	Use
`void println()`	Terminates the current line by writing the line separator string
`void println(boolean x)`	Prints a Boolean value and then terminates the line
`void println(char x)`	Prints a character and then terminates the line
`void println(char[] x)`	Prints an array of characters and then terminates the line
`void println(double x)`	Prints a `double` and then terminates the line
`void println(float x)`	Prints a `float` and then terminates the line
`void println(int x)`	Prints an `int` and then terminates the line
`void println(long x)`	Prints a `long` and then terminates the line
`void println(Object x)`	Prints an object and then terminates the line
`void println(String x)`	Prints a string and then terminates the line

ORACLE

Many methods in the Java API are overloaded, including the `System.out.println` method. The table in the slide shows all of the variations of the `println` method.

Quiz

Which method corresponds to the following method call?

```
myPerson.printValues(100, 147.7F, "lavender");
```

a. `public void printValues (int pantSize, float ageInYears)`

b. `public void printValues (pantSize, float ageInYears, favoriteColor)`

c. `public void printValues (int pantSize, float ageInYears, String favoriteColor)`

d. `public void printValues (float ageInYears, String favoriteColor, int pantSize)`

Answer: c

Summary

Methods:

- Make programs more readable and easier to maintain
- Make development and maintenance quicker
- Are central to reusable software
- Allow separate objects to communicate and to distribute the work performed by the program

Practice 10-1: Writing a Method with Arguments and Return Values

This practice covers the following topics:

- Creating a class to order more than one shirt
- Displaying the total order value of the shirts

Challenge Practice 10-2: Writing a Class Containing an Overloaded Method

This practice covers writing a Customer class with an overloaded method called `setCustomerInfo()`.

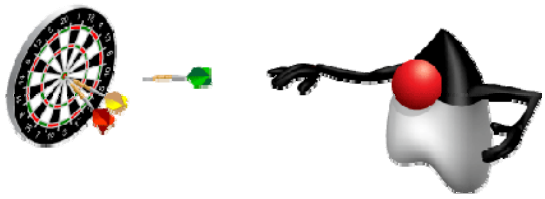
11

Using Encapsulation and Constructors

Objectives

After completing this lesson, you should be able to:

- Create a constructor
- Use access modifiers
- Describe the purpose of encapsulation
- Implement encapsulation in a class



ORACLE

Topics

- Encapsulation
- Constructors

Overview

- Encapsulation means hiding object fields by making all fields private:
  - Use getter and setter methods.
  - In setter methods, use code to check that values are valid.
- Encapsulation mandates programming to the interface:
  - Data type of the field is irrelevant to the caller method.
  - Class can be changed as long as interface remains same.
- Encapsulation encourages good OO design.

The `public` Modifier

```
public class Elevator {
   public boolean doorOpen=false;
   public int currentFloor = 1;
   public final int TOP_FLOOR = 10;
   public final int MIN_FLOOR = 1;

   ... < code omitted > ...

 public void goUp() {
    if (currentFloor == TOP_FLOOR) {
      System.out.println("Cannot go up further!");
    }
    if (currentFloor < TOP_FLOOR) {
      currentFloor++;
      System.out.println("Floor: " + currentFloor);
    }
  }
}
```


The code in the slide shows the `goUp()` method and the `currentFloor` field. It's the corresponding method to the `goDown()` method previously discussed, and protects the elevator from trying to go above the top floor.

But the way the code is shown here indicates a problem. The `goUp()` method can be circumvented; there is nothing to stop the `currentFloor` field from being modified directly.

Dangers of Accessing a `public` field

```
Elevator theElevator = new Elevator();

theElevator.currentFloor = 15;
```

Could cause a problem!

The `private` Modifier

```
public class Elevator {
   private boolean doorOpen=false;
   private int currentFloor = 1;
   private final int TOP_FLOOR = 10;
   private final int MIN_FLOOR = 1;

   ... < code omitted > ...

 public void goUp() {
    if (currentFloor == TOP_FLOOR) {
      System.out.println("Cannot go up further!");
    }
    if (currentFloor < TOP_FLOOR) {
      currentFloor++;
      System.out.println("Floor: " + currentFloor);
    }
  }
}
```

None of these fields can now be accessed from another class using dot notation.

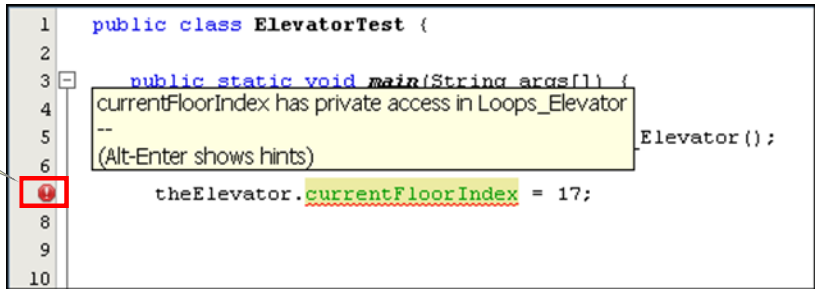
In the example shown, the fields have all been made private. Now they cannot be accessed from a caller method that is outside this class. So any calling method that wants to control the floor that the elevator will go to must do so through its public methods.

Dangers of Accessing a `public` field

```
Elevator theElevator = new Elevator();

theElevator.currentFloor = 15;
```

not permitted

NetBeans will show an error. You can get an explanation if you place your cursor here.



The `private` Modifier on Methods

```
public class Elevator {
   ... < code omitted > ...

  private void setFloor() {
     int desiredFloor = 5;
     while (  currentFloor != desiredFloor  ){
        if (currentFloor < desiredFloor) {
           goUp();
        } else {
           goDown();
        }
     }
  }

  public void requestFloor(int desiredFloor) {
     ... < contains code to add requested floor to a queue > ...
  }
```

Should this method be private?

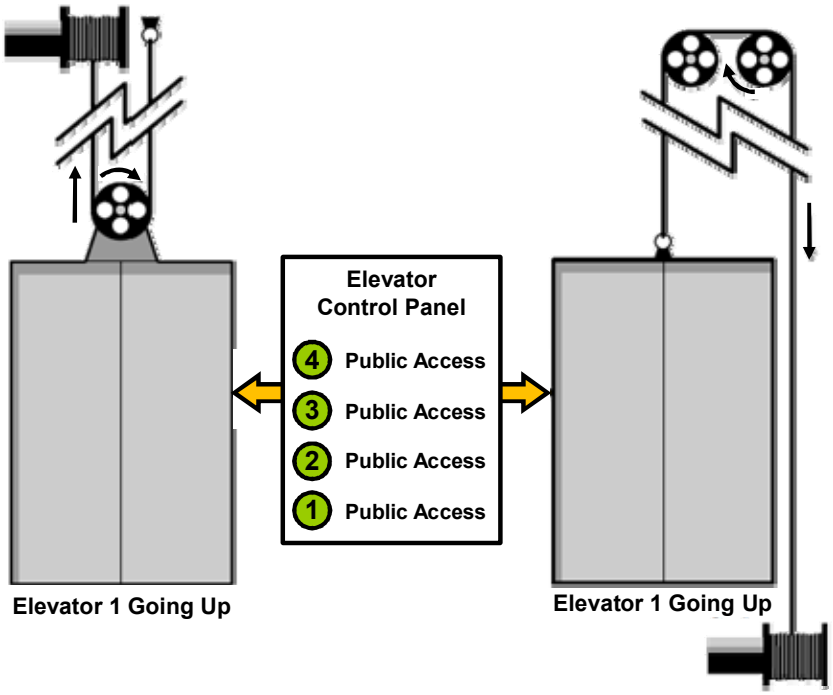
Remember the `setFloor()` method? Methods are declared with a modifier just like fields. Can you think of a reason why this method might be best declared with a private modifier?

Well, if the elevator works like most elevators do, the controls operated by the general public (either the button to call an elevator, or the button to request a floor) do not directly affect the elevator.

Instead, a user presses a button, say, a request for an elevator to go to the fifth floor. The elevator does not respond immediately to the request, but puts the request in a queue and then eventually, perhaps after bringing users already on the elevator down to the first floor, goes to the fifth floor.

It may be that the only public method needs to be `requestFloor()`, at least for the software that controls the buttons used by the general public.

Interface and Implementation



When classes are encapsulated, other objects interact with only a few parts (methods) of every other class.

In the example of the elevator, the control program that is triggered by the buttons can only call the `requestFloor()` method of `Elevator`. And, as long as `Elevator` implements this method, it doesn't matter exactly how it is implemented. The method could store requests in a binary array where setting an element to true indicates that there is a request on the floor with that index. Or an ArrayList could be used to store the numbers of the floors requested.

There might also be a `moveElevator()` method that is triggered by something, perhaps by the doors closing. Again, as long as this method `moveElevator()` is implemented, its implementation can be changed to change the way in which the elevator responds to requests coming in at the same time from different floors.

Get and Set Methods

```
public class Shirt {
   private int shirtID = 0; // Default ID for the shirt
   private String description = "-description required-"; // default
   // The color codes are R=Red, B=Blue, G=Green, U=Unset
   private char colorCode = 'U';
   private double price = 0.0; // Default price for all items

   public char getColorCode() {
     return colorCode;
   }
   public void setColorCode(char newCode) {
     colorCode = newCode;
   }
   // Additional get and set methods for shirtID, description,
   // and price would follow

} // end of class
```

ORACLE

If you make attributes private, how can another object access them? One object can access the private attributes of a second object if the second object provides public methods for each of the operations that are to be performed on the value of an attribute.

For example, it is recommended that all fields of a class should be private, and those that need to be accessed should have public methods for setting and getting their values.

This ensures that, at some future time, the actual field type itself could be changed, if that were advantageous. Or the getter or setter methods could be modified to control how the value could be changed, in the same way you wrote code to ensure that the `currentFloor` field of the elevator could not be set to an invalid value.

Using Setter and Getter Methods

```
public class ShirtTest {
   public static void main (String args[]) {
   Shirt theShirt = new Shirt();
   char colorCode;

   // Set a valid colorCode
   theShirt.setColorCode('R');
   colorCode = theShirt.getColorCode();
   // The ShirtTest class can set and get a valid colorCode
   System.out.println("Color Code: " + colorCode);

   // Set an invalid color code
   theShirt.setColorCode('Z');   <-- not a valid color code
   colorCode = theShirt.getColorCode();
   // The ShirtTest class can set and get an invalid colorCode
   System.out.println("Color Code: " + colorCode);
}
```

ORACLE

Though the code for the `Shirt` class is syntactically correct, the `setcolorCode` method does not contain any logic to ensure that the correct values are set.

The code example in the slide successfully sets an invalid color code in the Shirt object.

However, because `ShirtTest` accesses a private field on `Shirt` via a setter method, `Shirt` can now be recoded without modifying any of the classes that depend on it.

Setter Method with Checking

```
public void setColorCode(char newCode) {
   switch (newCode) {
      case 'R':
      case 'G':
      case 'B':
      colorCode = newCode;
      break;
      default:
      System.out.println("Invalid colorCode. Use R, G, or B");
   }
}
```


In the slide is another version of the `Shirt` class. However, in this class, before setting the value, the setter method ensures that the value is valid. If it is not valid, the `colorCode` field remains unchanged and an error message is printed.

Using Setter and Getter Methods

```
public class ShirtTest {
   public static void main (String args[]) {
   Shirt theShirt = new Shirt();
   System.out.println("Color Code: " + theShirt.getColorCode());

   // Try to set an invalid color code
   Shirt1.setColorCode('Z');
   System.out.println("Color Code: " + theShirt.getColorCode());
}
```

not a valid color code

Output:

```
Color Code: U
Invalid colorCode. Use R, G, or B
Color Code: U
```

Before call to `setColorCode()` – shows default value

call to `setColorCode` prints error message

`colorCode` not modified by invalid argument passed to `setColorCode()`

Topics

- Encapsulation
- Constructors

Initializing a Shirt Object

```
public class ShirtTest {
   public static void main (String args[]) {
   Shirt theShirt = new Shirt();

   // Set values for the Shirt
   theShirt.setColorCode('R');
   theShirt.setDescription("Outdoors shirt");
   theShirt.price(39.99);

}
```

ORACLE

Assuming you now have setters for all the private fields of `Shirt`, you could now instantiate and initialize a Shirt object by instantiating it and then setting the various fields through the setter methods.

However, Java provides a much more convenient way to instantiate and initialize an object by using a special method called a constructor.

Constructors

Constructors are method-like structures in a class:

- They have the same name as the class.
- They are usually used to initialize fields in an object.
- They can be overloaded.

Creating Constructors

Syntax:

```
[modifiers] class ClassName {

    [modifiers] ClassName([arguments]) {
      code_block
    }
}
```


`[modifiers]` represent several unique Java technology keywords that can modify the way constructors are accessed. Modifiers are optional (indicated by the square brackets).

`ClassName` is the name of the class and the name of the constructor method. The `name` of the constructor must be the same as the `ClassName` in the class declaration.

`[arguments]` represent one or more optional arguments passed to the constructor.

The `code_block` represents one or more optional lines of code for the constructor.

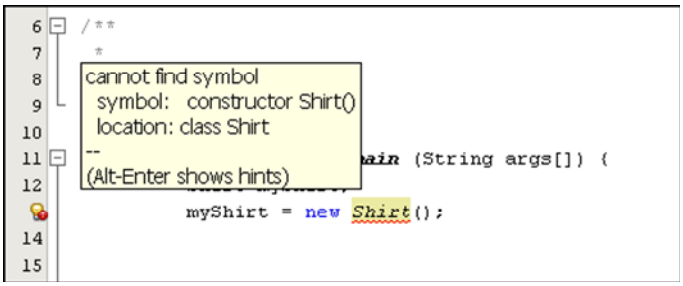
Creating Constructors

```
public class Shirt {
  public int shirtID = 0; // Default ID for the shirt
  public String description = "-description required-"; // default
  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = 'U';
  public double price = 0.0; // Default price all items

 // This constructor takes one argument
  public Shirt(char colorCode ) {
      setColorCode(colorCode);
  }
```


The Shirt example shown in the slide has a constructor that accepts a `char` value to initialize the color code for this object. Because `setColorCode()` ensures that an invalid code cannot be set, the constructor can just call this method.

Initializing a Shirt Object by Using a Constructor

```
public class ShirtTest {
    public static void main (String args[]) {
    Shirt theShirt = new Shirt('G');

    theShirt.display();
}
```



As you would expect, passing a valid color code to the Shirt constructor creates a new Shirt object, and calling `display()` results in the following output:

```
Item ID: 0
Item description:-description required-
Color Code: G
Item price: 0.0
```

However, look at the message you get in NetBeans if you try to call the Shirt constructor with no arguments, the way you have been doing up until now.

The reason for the problem is that if there is no explicit constructor in a class, Java assumes you want to be able to instantiate the class and gives you a default no-argument constructor; otherwise how could you instantiate the class?

But if you create one constructor, Java assumes that you might want that to be the only constructor, and no longer provides a default no-argument implementation.

Default Constructor

```
public class Shirt {
  ... < declarations for field omitted > ...

  // Default constructor
  public Shirt() {
     // You could add some default processing here
  }
  // This constructor takes one argument
  public Shirt(char colorCode ) {
      setColorCode(colorCode);
  }
  public Shirt(char colorCode, double price) {

      this(colorCode);
      setPrice(price);
  }
```

Chaining the constructors

The code in the slide shows three overloaded constructors:

- A default constructor with no parameters
- A constructor with one parameter (a `char`)
- A constructor with two parameters (a `char` and a `double`)

This third constructor sets both the `colorCode` field and the `price` field. Notice, however, that the syntax where it sets the `colorCode` field is one you haven't seen yet. It would be quite possible to set `colorCode` with a simple call to `setColorCode()` just as the previous constructor does, but there is another option, as shown here.

Imagine that the second constructor has a lot of quite complex code associated with setting the `colorCode`. If that were the case, you would also not want to duplicate the code in the third constructor and so you would chain the constructors by calling the second constructor in the first line of the third constructor, using the following syntax:

```
this(argument);
```

`this` is a special keyword that is a reference to the current object.

Quiz

What is the default constructor for the following class?

```
public class Penny {
    String name = "lane";
}
```

a. `public Penny(String name)`
b. `public Penny()`
c. `class()`
d. `String()`

Answer: b

Summary

Encapsulation protects data:

- By making all fields private
  - Use getter and setter methods.
  - In setter methods, use code to check that values are valid.
- By mandating programming to the interface
  - Data type of the field is irrelevant to the caller method.
  - Class can be changed as long as interface remains same.
- By encouraging good OO design

Practice 11-1: Implementing Encapsulation in a Class

This practice covers the following topics:

- Implementing encapsulation in a class
- Accessing encapsulated attributes of a class

ORACLE

Challenge Practice 11-2: Adding Validation to the `DateThree` Class

This practice covers adding a setter method that performs validation.

Practice 11-3:
Creating Constructors to Initialize Objects

This practice covers creating constructors to initialize objects.

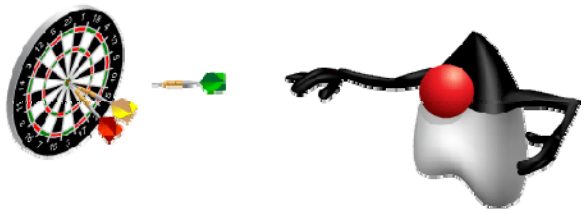
12

Using Advanced Object-Oriented Concepts

Objectives

After completing this lesson, you should be able to:

- Describe inheritance
- Describe polymorphism
- Test superclass and subclass relations
- Create a subclass
- Use abstract classes and interfaces

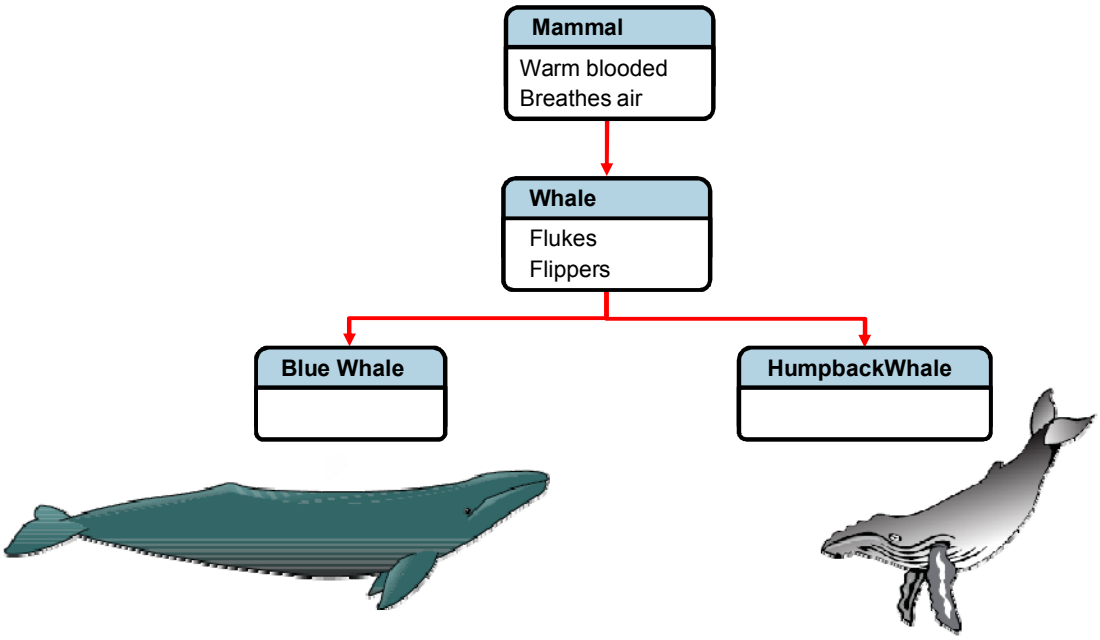


ORACLE

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Polymorphism and overriding methods
- Interfaces
- The Object object

Class Hierarchies



Inheritance results in a class hierarchy of Java technology classes similar to the taxonomies found in biology, such as “Blue Whale is a subclass of Whale.”

The diagram in the slide illustrates a hierarchy for whales. “Warm blooded” is an attribute of the Mammal superclass. The phrase “breathes air” represents some operation that is also a part of the Mammal superclass. Flukes and flippers are attributes that are specific to Whale, which is a subclass of mammal.

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Polymorphism and overriding methods
- Interfaces
- The Object object

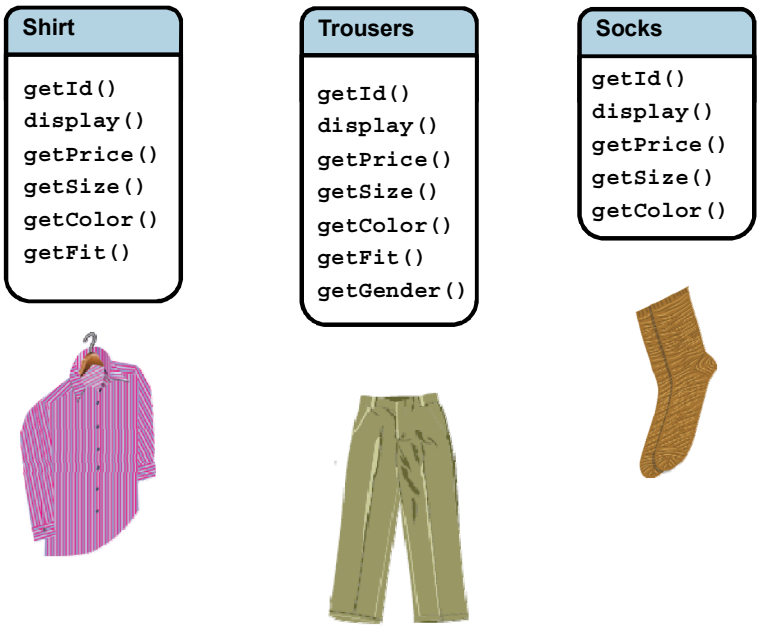
Common Behaviors

Shirt	Trousers
getId() getPrice() getSize() getColor() getFit()	getId() getPrice() getSize() getColor() getFit() getGender()
setId() setPrice() setSize() setColor() setFit()	setId() setPrice() setSize() setColor() setFit() setGender()
display()	display()

ORACLE

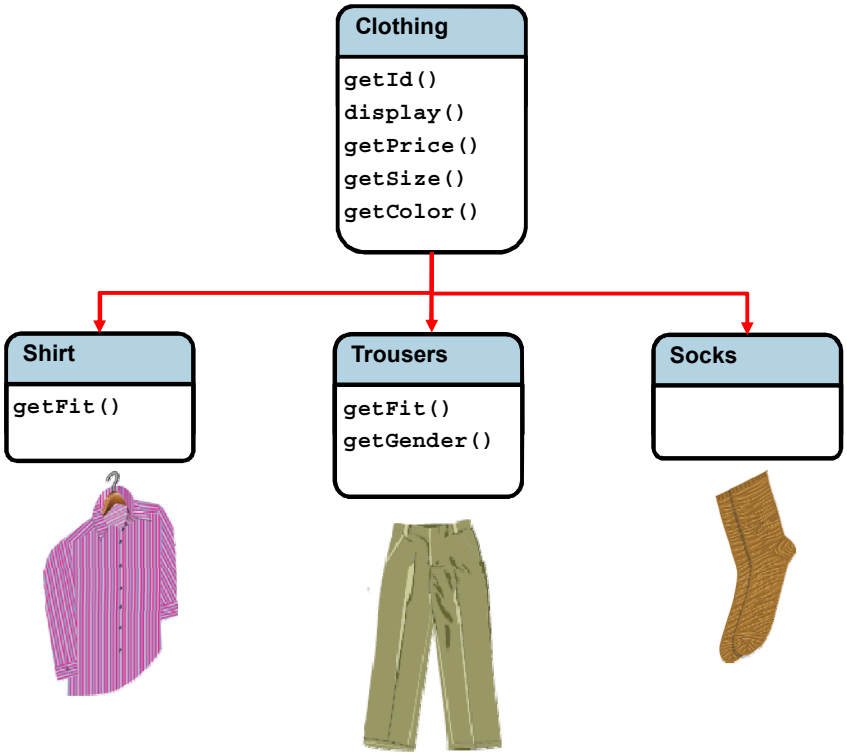
The diagram shows a set of behaviors for the Shirt class and for a new class, the Trousers class. The classes are shown fully encapsulated so that all field values are only accessible through setter and getter methods. Notice how both classes use many of the same methods; this may result in code duplication, making maintenance and further expansion more difficult and error prone.

Code Duplication



If Duke's Choice decides to add a third item, socks, as well as trousers and shirts, you may find even greater code duplication. The diagram in the slide shows only the getter methods for accessing the properties of the new objects.

Inheritance



You can eliminate the need to duplicate this information in every class by implementing inheritance. Inheritance enables programmers to put common members (fields and methods) in one class (the superclass) and have other classes (the subclasses) inherit these common members from this new class.

An object instantiated from a subclass behaves as if the fields and methods of the subclass were in the object. For example, the Trousers class can be instantiated and have the display() method called even though the Trousers class does not contain a display() method; it is inherited from the Clothing class.

Overriding Superclass Methods

Methods that exist in the superclass can be:

- Not implemented in the subclass
  - The method declared in the superclass will be used at runtime.
- Implemented in the subclass
  - The method declared in the subclass will be used at runtime.

Subclasses may implement methods that already have implementations in the superclass. In this case the method implementations in the subclass are said to override the method implementation from the superclass. For example, although the colorCode field (and its accessor methods) is in the superclass, the color choices may be different in each subclass. So it may be necessary to override the get and set methods for this field in the individual subclasses.

The Clothing Superclass: 1

```
public class Clothing  {
  // Fields
  private int itemID = 0; // Default ID for all clothing items
  private String description = "-description required-"; // default
  private char colorCode = 'U'; //'U' is Unset
  private double price = 0.0; // Default price for all items

  // Constructor
  public Clothing(int itemID, String description, char colorCode,
    double price) {
    this.itemID = itemID;
    this.description = description;
    this.colorCode = colorCode;
    this.price = price; }
```

ORACLE

The code listed in the slide shows the fields and the constructor for the Clothing superclass.

The Clothing Superclass: 2

```
public void display() {
   System.out.println("Item ID: " + getItemID());
   System.out.println("Item description: " + description);
   System.out.println("Item price: " + getPrice());
   System.out.println("Color code: " + getColorCode());
 } // end of display method
 public String getDescription(){
     return description;
 }
 public double getPrice() {
     return price;
 }
 public int getItemID() {
     return itemID;
 }
```

ORACLE

The code listed in the slide shows methods for the Clothing superclass.

The Clothing Superclass: 3

```
public char getColorCode() {
    return colorCode;
}
public void setItemID(int itemID) {
    this.itemID = itemID;
}
public void setDescription(String description) {
    this.description = description;
}
public void setColorCode(char colorCode) {
    this.colorCode = colorCode;
}
public void setPrice(double price) {
    this.price = price;
}
```


The code listed in the slide shows the remaining methods of the Clothing superclass.

Declaring a Subclass

Syntax:

```
[class_modifier] class class_identifier extends superclass_identifier
```

Declaring a Subclass (`extends`, `super`, and `this` keywords)

```
public class Shirt extends Clothing {

  private char fit = 'U'; //'U' is Unset, other codes 'S', 'M', or 'L'

  public Shirt(int itemID, String description, char colorCode,
               double price, char fit) {
     super(itemID, description, colorCode, price);

     this.fit = fit;
  }

  public char getFit() {
      return fit;
  }
  public void setFit(char fit) {
      this.fit = fit;
  }
```

Ensures that Shirt inherits members of Clothing

`super` is a reference to methods and attributes of the superclass

`this` is a reference to this object

ORACLE

The code listed in the slide shows the code of the `Shirt` subclass. The code in this class declares attributes and methods that are unique to this class. Attributes and methods that are common with the `Clothing` class are inherited and do not need to be declared.

It also includes two useful keywords and shows a common way of implementing constructors in a subclass.

`super` refers to the superclass. Even if a method of the superclass has been overridden in the subclass, using the `super` keyword allows you to invoke the method of the superclass. In the example in the slide, it is being used to invoke the constructor on the superclass. By using this technique, the constructor on the superclass can be invoked to set all the common attributes of the object being constructed. Then, as in the example here, additional attributes can be set in following statements.

The only additional attribute that `Shirt` has is the `fit` attribute, and it is set after the invocation of the superclass constructor. Note the use of the `this` keyword. In contrast to the `super` keyword, `this` is a reference to the object of this class. It is not necessary to use it in the example in the slide, but it is common to do so in constructors to help make the code more readable.

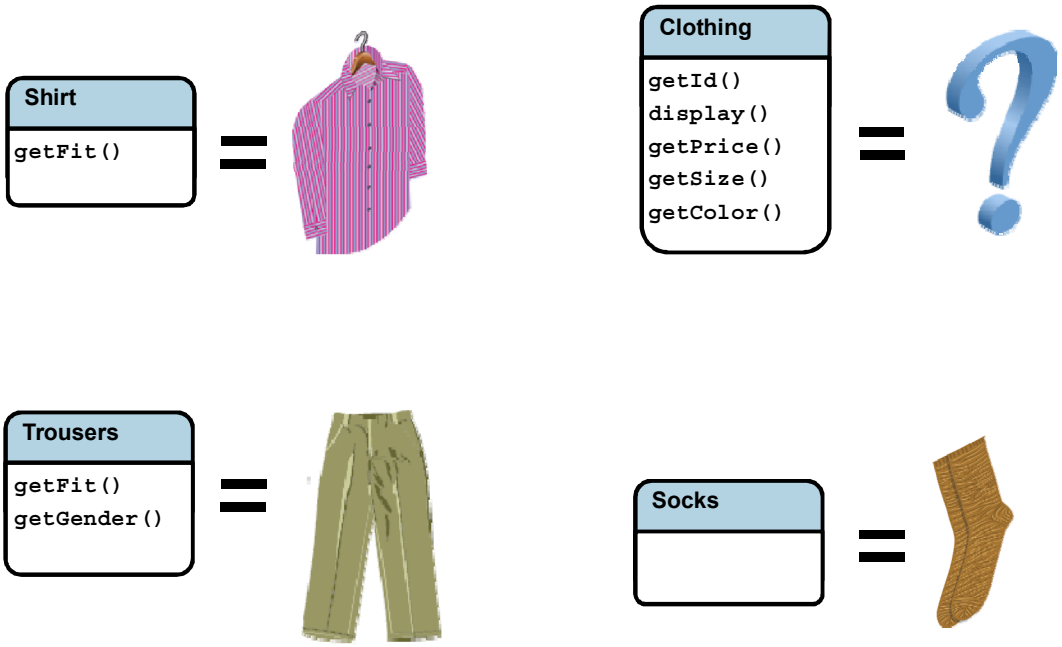
Declaring a Subclass: 2

```
  //This method overrides display in the Clothing superclass
  public void display() {
    System.out.println("Shirt ID: " + getItemID());
    System.out.println("Shirt description: " + description);
    System.out.println("Shirt price: " + getPrice());
    System.out.println("Color code: " + getColorCode());
    System.out.println("Fit: " + getFit());
  } // end of display method

  // This method overrides the methods in the superclass
  public void setColorCode(char colorCode) {

      ... include code here to check that correct codes used ...

      this.colorCode = colorCode;
  }
}
```


Notice that the method `display()` overrides the `display()` method of the superclass and is more specific to the Shirt class.

Likewise the method `setColorCode()` overrides the `setColorCode()` method of the superclass to check that a valid value is being used for this class. (The code is not shown here, but remember that this is one of the advantages of encapsulating fields, as discussed in the lesson titled “Using Encapsulation and Constructors.”)

Abstract Classes



Sometimes a superclass makes sense as an object, and sometimes it does not. Duke's Choice carries shirts, socks, and trousers, but it does not have an individual item called a “clothing.” Also, in the application, the superclass Clothing may declare some methods that may be required in each subclass (and thus can be in the superclass) but cannot really be implemented in the superclass.

The Abstract Clothing Superclass: 1

```
public abstract class Clothing  {
  // Fields
  private int itemID = 0; // Default ID for all clothing items
  private String description = "-description required-"; // default
  private char colorCode = 'U'; //
  private double price = 0.0; //  all items

  // Constructor
  public Clothing(int itemID, String description, char colorCode,
    double price, int quantityInStock) {
    this.itemID = itemID;
    this.description = description;
    this.colorCode = colorCode;
    this.price = price;
  }
```

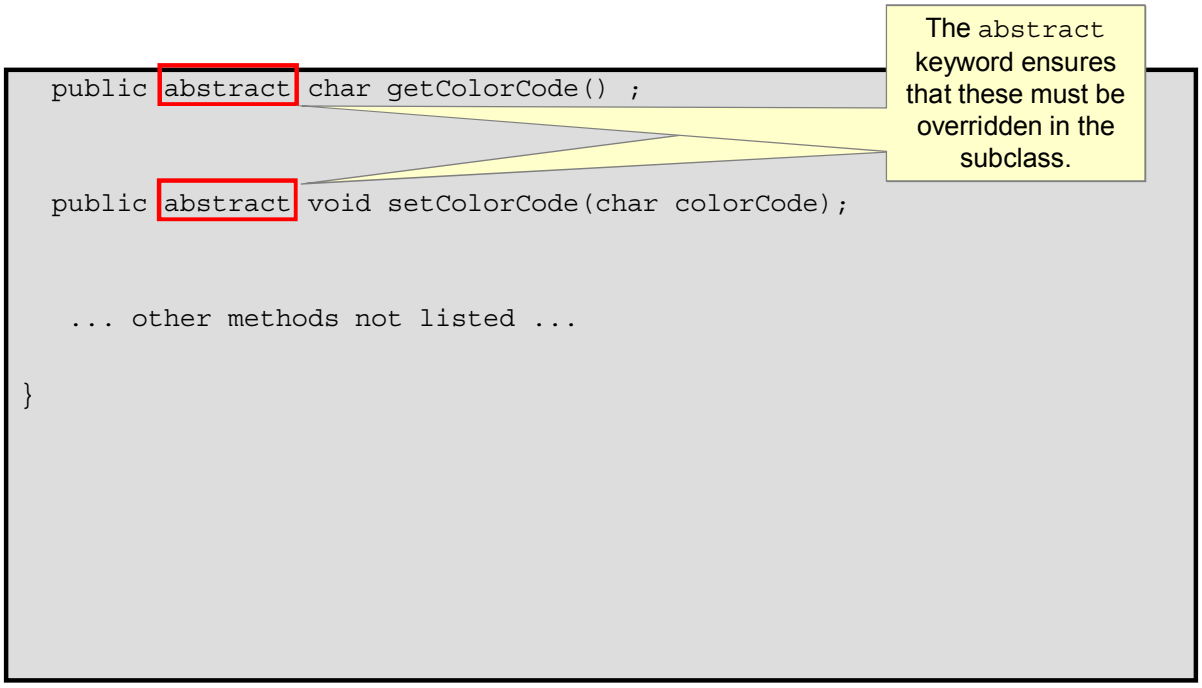
The `abstract` keyword ensures that the class cannot be instantiated.

Abstraction refers to creating classes that are general and may contain methods without particular implementation or method body code.

A example of an abstract class is the Clothing class as coded on this slide and the next. Clothing is an abstract concept (you usually do not go to a store and say, “I want to buy a clothing item”) that can refer to anything.

However, all clothing items have some similar characteristics in the context of an order entry system, such as an ID or a method to display information about the item. Classes that are generic and cannot be fully defined, such as an Item class, are referred to as abstract classes. Classes that extend an abstract class must implement the empty methods of the abstract class with code specific to the subclass. You will want to spend time on your analysis and design to make sure that your solution has enough abstraction to ensure flexibility.

The Abstract Clothing Superclass: 2



The get and set methods for the colorCode field are abstract to ensure that they are implemented appropriately in each subclass.

Note that the `Shirt` subclass shown previously will compile correctly as a subclass of this abstract class, as it already has implementations of these two methods. But if the implementations of `getColorCode()` and `setColorCode()` are removed from the `Shirt` subclass, the compile will fail, as abstract methods in the superclass must be implemented in the subclass.

Superclass and Subclass Relationships

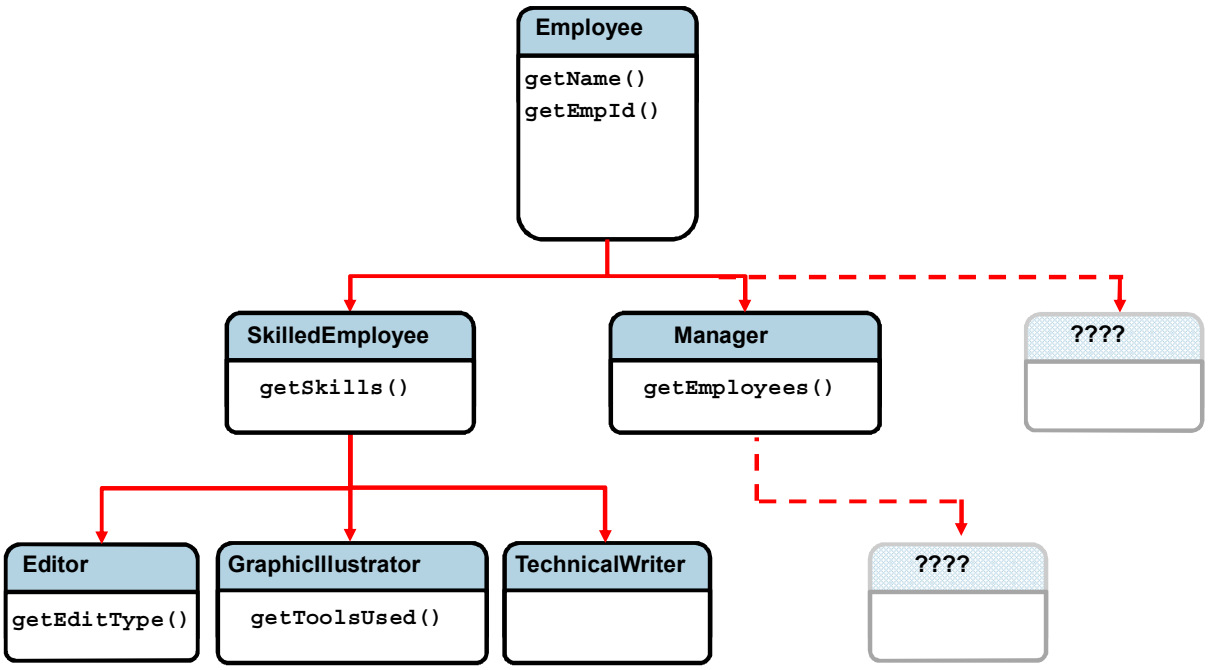
It is very important to consider the best use of inheritance

- Use inheritance only when it is completely valid or unavoidable.
- Check appropriateness with the "*is a*" phrase.
  - The phrase "a Shirt is a piece of Clothing" expresses a valid inheritance link.
  - The phrase "a Hat is a Sock" expresses an invalid inheritance link.

ORACLE

In the examples in this course, shirts, trousers, hats, and socks are all types of clothing. So Clothing is a good name for the superclass to these subclasses (types) of clothing.

Another Inheritance Example



Here's an example of another set of superclasses and subclasses. In this case there are more than two levels. The base superclass is Employee, and Employee currently has two subclasses. Of course, one of the big advantages of inheritance is that it is easy at any future time to create a new class that extends Employee , and that class will inherit all the functionality that Employee has.

One of the Employee subclasses is SkilledEmployee, and the diagram shows that it has three subclasses of its own: Editor, GraphicIllustrator, and TechnicalWriter.

None of these classes are abstract. There is such a thing as an employee and some processes in an application using these classes may work with the Employee class.

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Polymorphism and overriding methods
- Interfaces
- The Object object

Superclass Reference Types

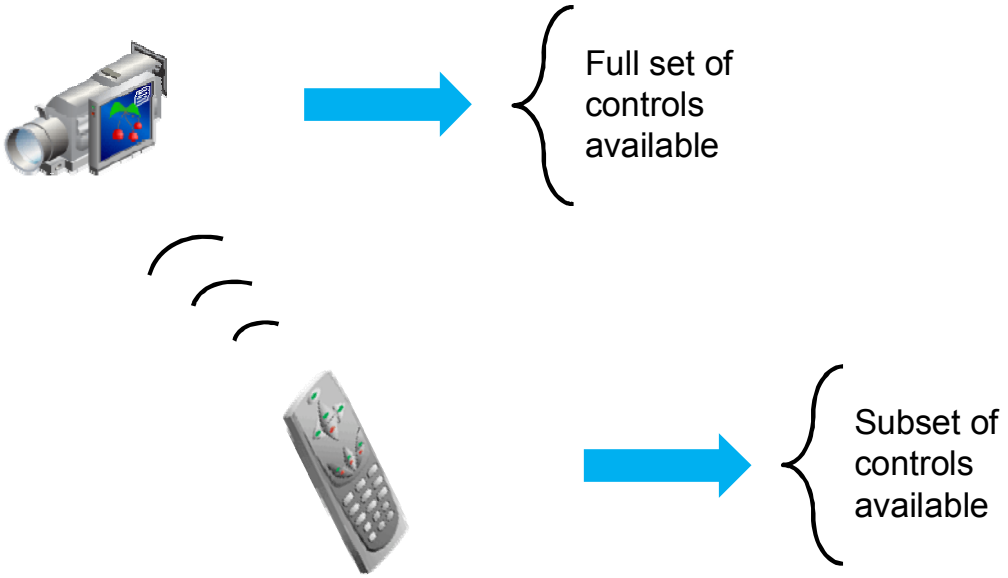
So far you have seen the class used as the reference type for the created object

- To use the Shirt class as the reference type for the Shirt object:
```
Shirt myShirt = new Shirt();
```
- But you can also use the superclass as the reference:
```
Clothing clothingItem1 = new Shirt();
Clothing clothingItem2 = new Trousers();
```

ORACLE

This ability to use not just the class itself, but any superclass of the class as its reference type is a very important feature of Java. In the example shown in the slide, notice that you can refer to both a Shirt object and a Trousers object with a Clothing reference. This means that a reference to a Shirt or Trousers object can be passed in to a method that requires a Clothing reference. Or a Clothing array can contain references to Shirt, Trousers, or Socks objects.

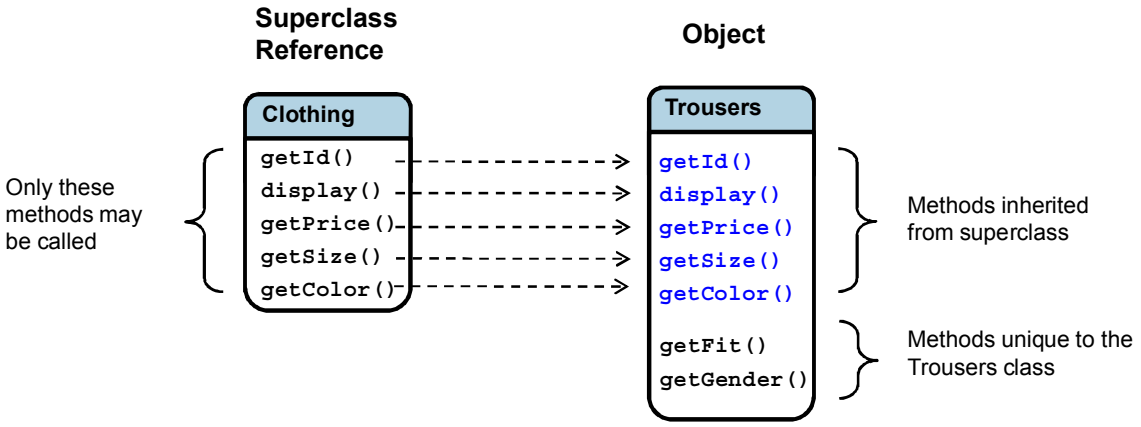
Access to Object Functionality



Accessing the methods of a class using a superclass reference is a little like accessing the controls of an electronic device using a remote control instead of the controls on the device itself. Often a device such as a video camera has a comprehensive set of controls for recording, playing, editing, and otherwise accessing every available function of the camera. This is a lot like using the class of the object as the reference type.

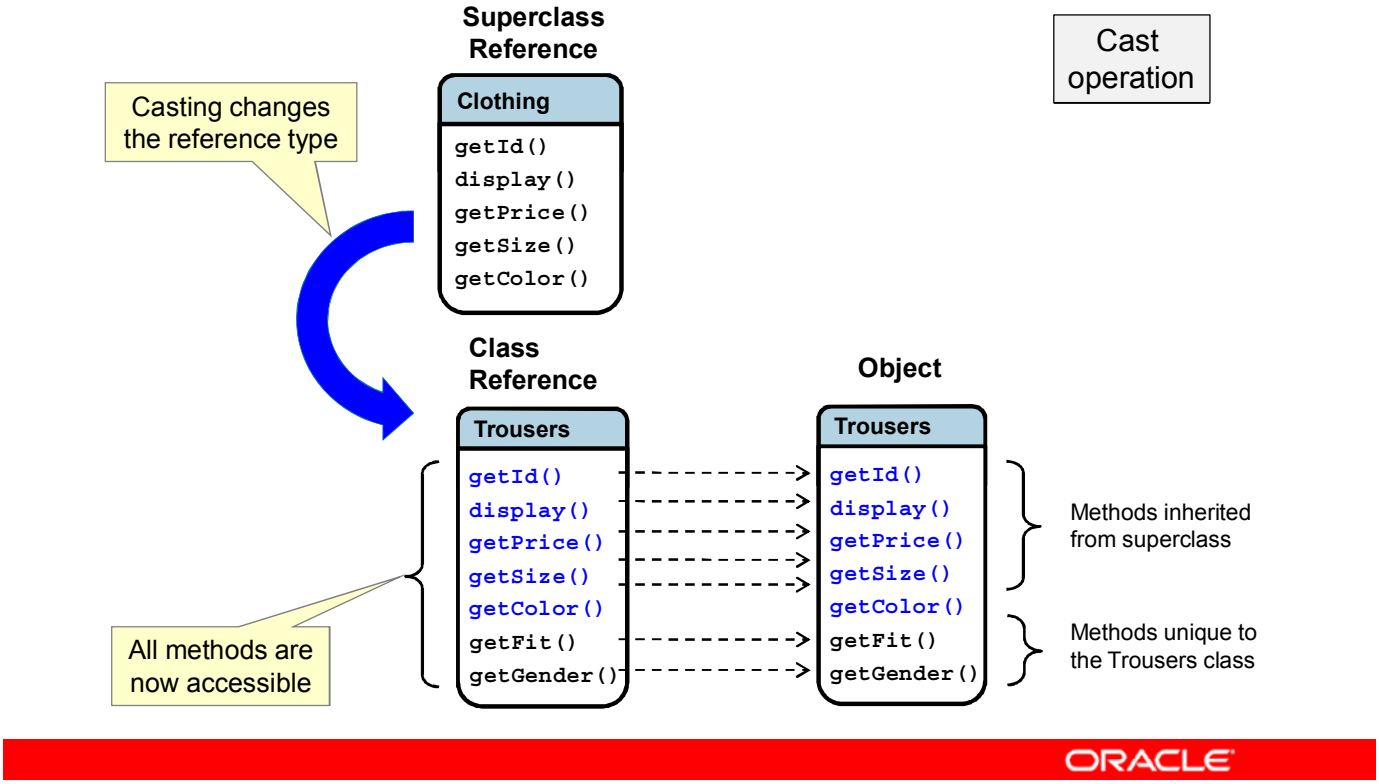
For some combinations of video camera and remote, the remote may give you exactly the same controls, and this can also be the case when using a superclass as reference for an object (the superclass gives you access to all the methods of the object; the object's class does not add any new methods). But it is often the case that the remote control does not have the full set of controls available on the camera itself, and again, this is common when using the superclass as reference. The superclass only has access to the methods of the class that are declared on the superclass even if the class has a number of other methods.

Accessing Class Methods from Superclass



Using a reference type `Clothing` will not allow access to the `getFit()` or `getGender()` methods of the `Trouser` object. Usually this is not a problem as you are most likely to be passing `Clothing` references to methods that do not require access to these methods. For example a `purchase()` method could receive a `Clothing` argument as it only needs access to the `getPrice()` method.

Casting the Reference Type



Given that a superclass may not have access to all the methods of the object it is referring to, how can you access those methods? The answer is that you can do so by replacing the superclass reference by:

- A reference that is the same type as the object
- An interface that declares the method(s) in question and is implemented by the class of the object

(Interfaces are covered in the next topic of this lesson.)

Casting

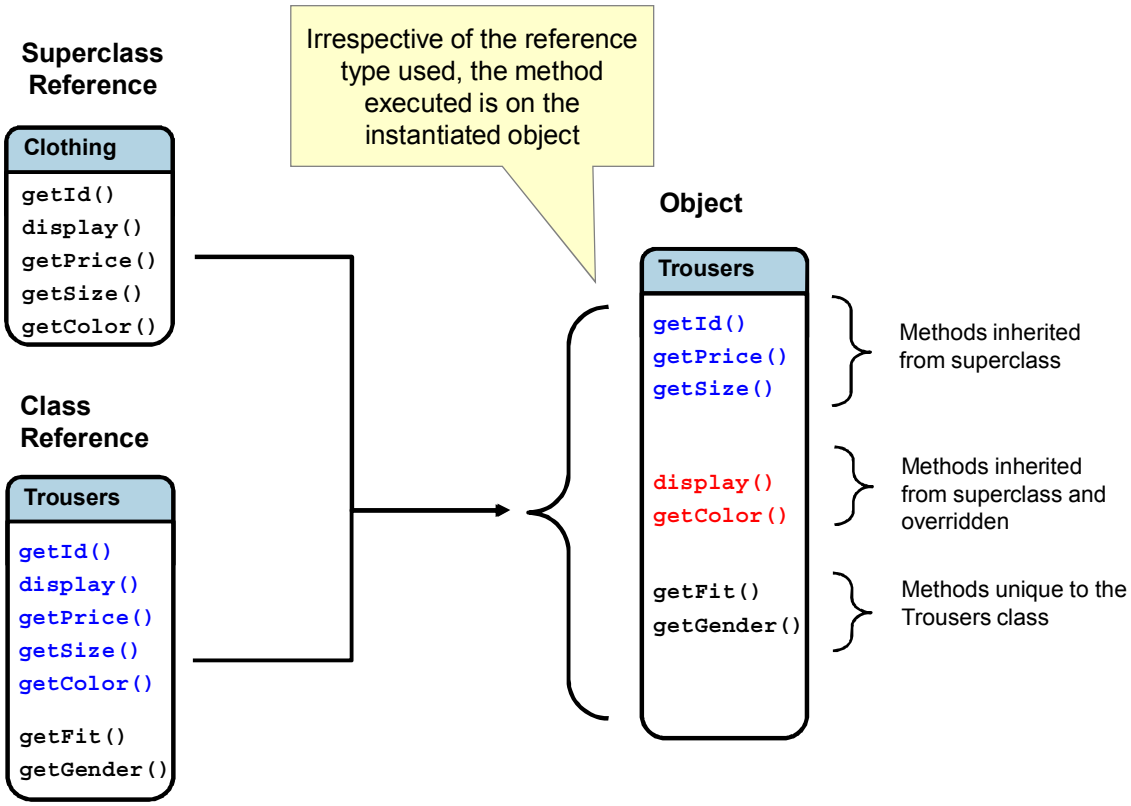
```
Clothing cl = new Trousers(123, "Dress Trousers", 'B', 17.00, 4, 'S');
cl.display();

//char fitCode = cl.getFit(); // This won't compile

char fitCode = ((Trousers)cl).getFit(); // This will compile
```

The parentheses around `cl` ensure that the cast applies to this reference.

The syntax for casting is the type to cast to in parentheses placed before the reference to be cast.

The code in this example shows a Clothing reference being cast to a `Trousers` reference to access the `getFit()` method, which is not accessible via the `Clothing` reference. Note that the inner parentheses around `Trousers` are part of the cast syntax, and the outer parentheses around `(Trousers)cl` are there to apply the cast to the `Clothing` type.

Polymorphic Method Calls



Quiz

How can you change the reference type of an object?

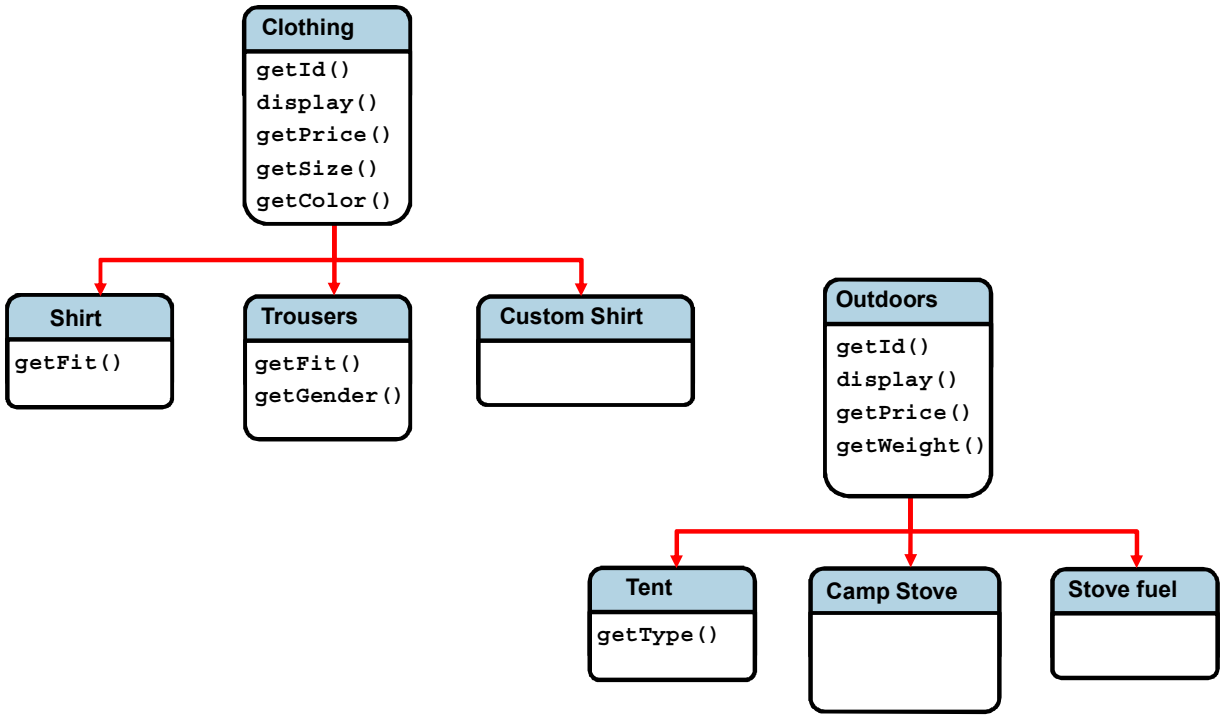
a. By calling `getReference()`
b. By casting
c. By declaring a new reference and assigning the object

Answer: b

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Polymorphism and overriding methods
- Interfaces
- The Object object

Multiple Hierarchies



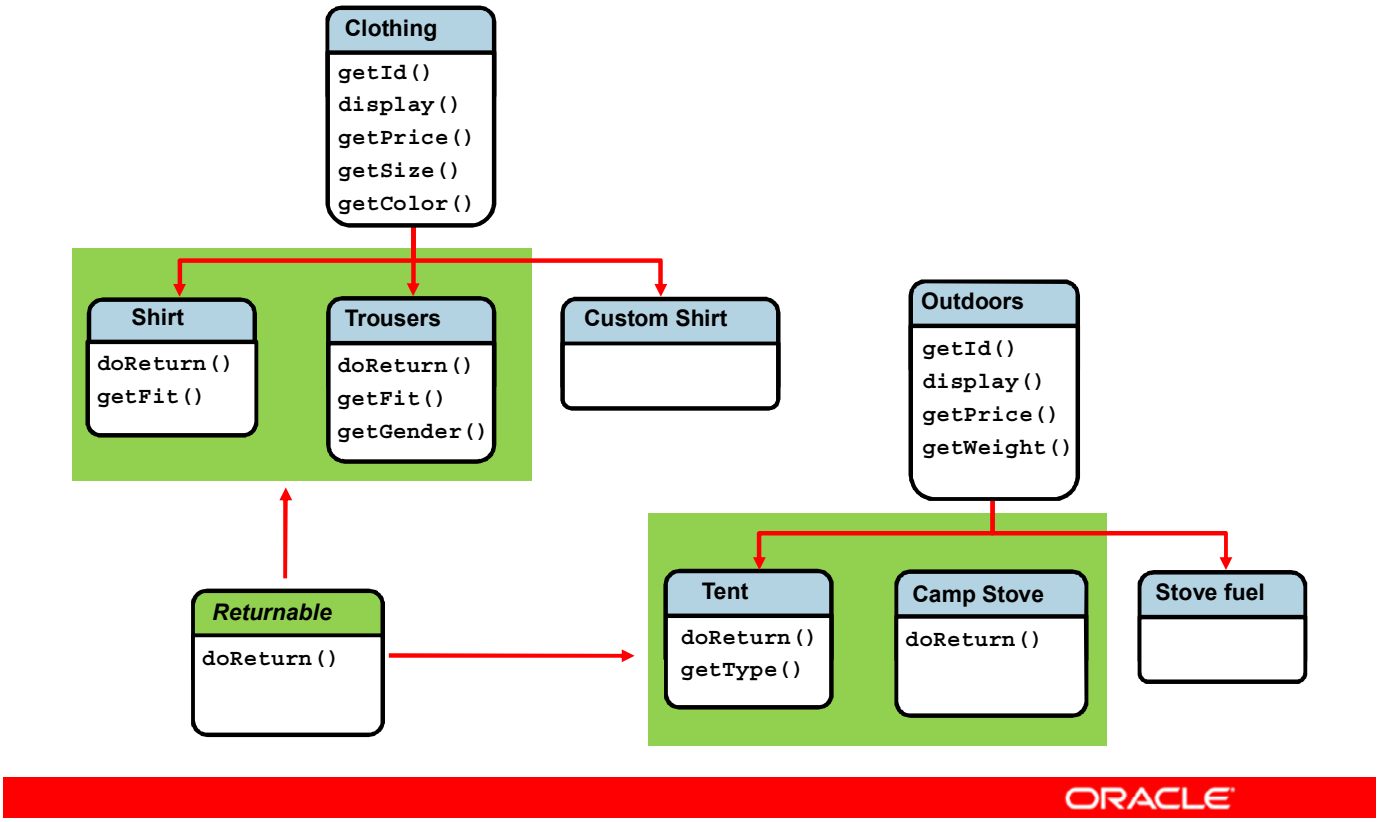
A more complex set of classes may have items in two quite different hierarchies. If Duke's Choice were to start selling outdoors gear, they may have a completely different superclass called Outdoors, with its own set of subclasses (for example, `getWeight()` as an Outdoors method).

In this scenario, there may be some classes from each hierarchy that have something in common. For example, the custom shirt item in Clothing is not returnable (since it's made by hand for a particular person); neither is the Stove fuel item in the Outdoors hierarchy. All other items are returnable.

How can this be modeled? Here are some things to consider:

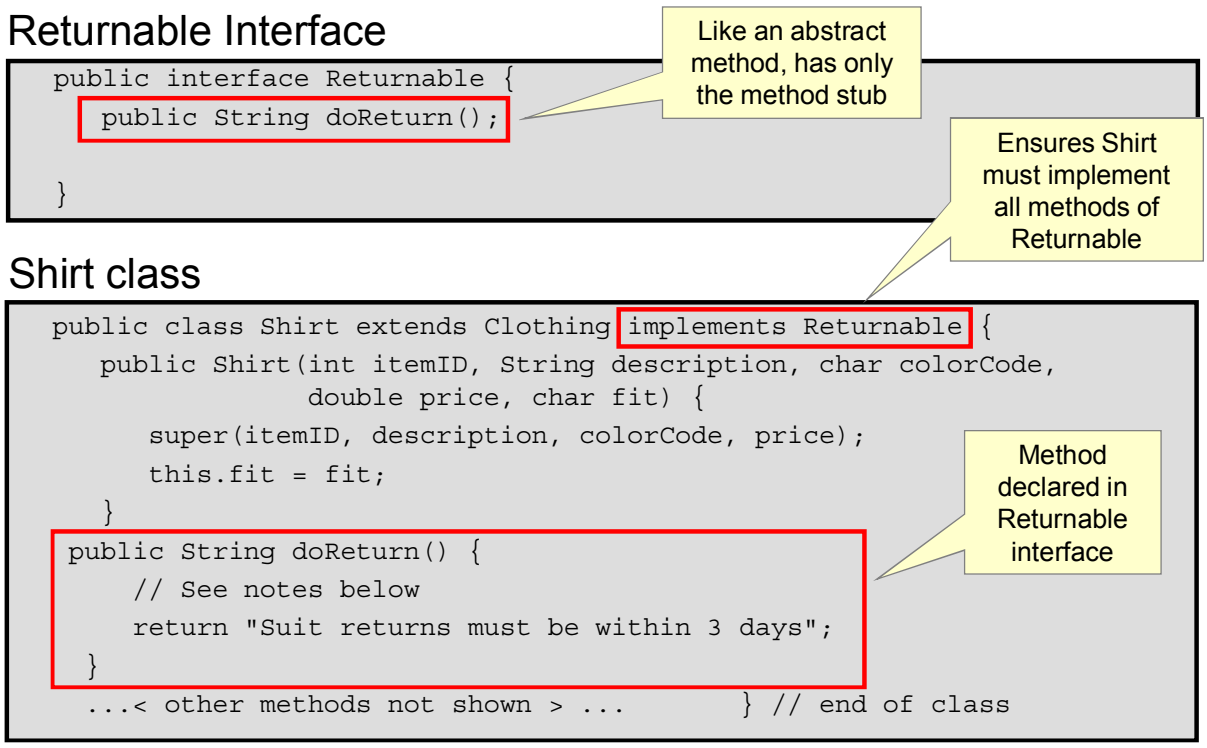
- A new superclass won't work because a class can only extend one superclass, and all items are currently extending either Outdoors or Clothing.
- A new field named returnable, added to every class, could be used to determine if an item can be returned. This is certainly possible but then there is no single reference type to pass to a method that initiates or processes a return.
- A special type called an Interface that can be implemented by any class. This Interface type can then be used to pass a reference of any class that implements it.

Interfaces



The diagram now shows all returnable items implementing the Returnable interface with its single method, `doReturn()`. Methods can only be declared in an interface, so each class that implements Returnable must implement `doReturn()` for itself. All returnable items could be passed to a `processReturns()` method of a Returns class and then have their `doReturn()` method called.

Implementing the Returnable Interface

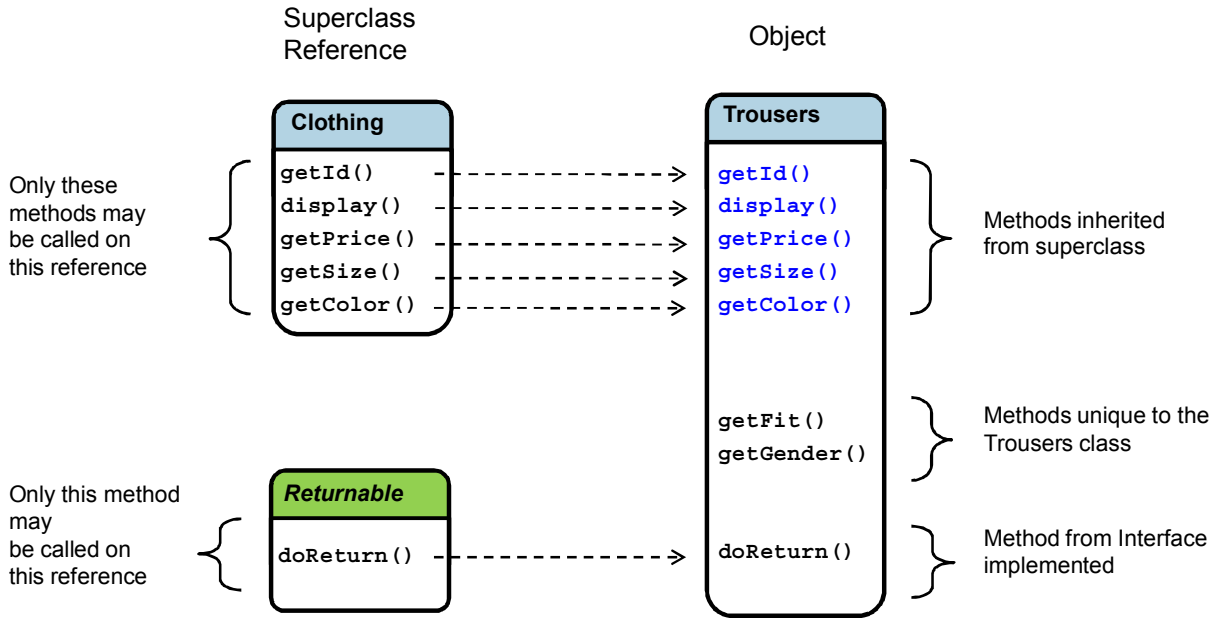


The code in this example shows the Returnable interface and the `Shirt` class. Only the constructor and the `doReturn()` methods are shown.

In this implementation, `Returnable` provides a marker to indicate that the item can be returned, and ensures that the developer of Shirt must implement the `doReturn()` method.

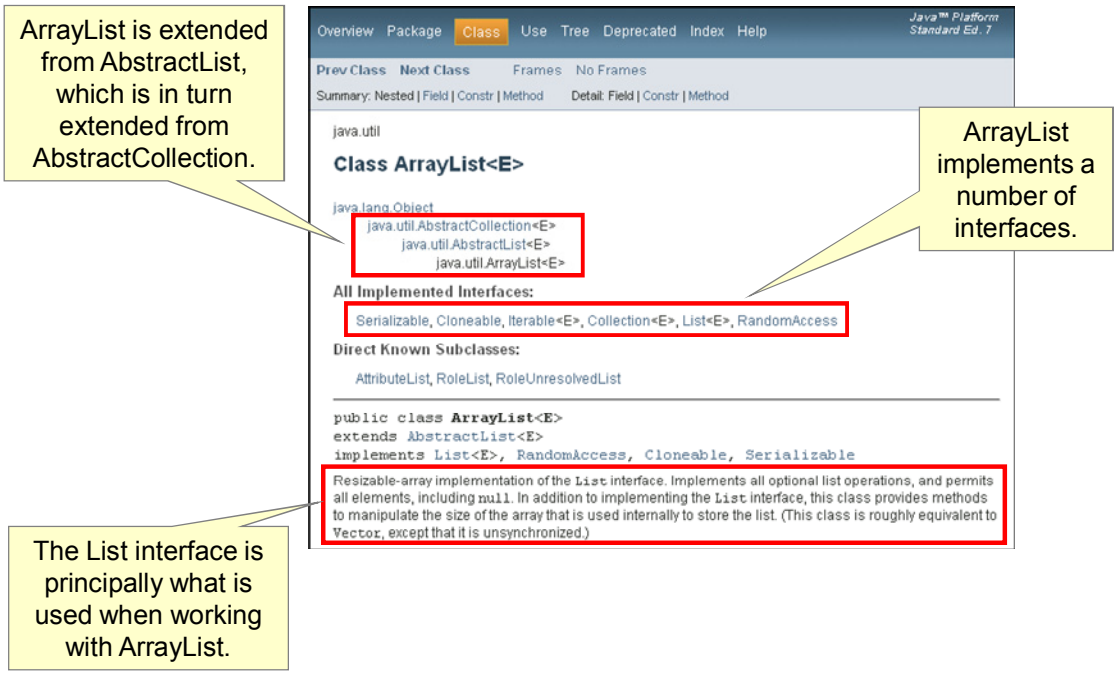
The `doReturn()` method returns a String describing the conditions for returning the item.

Access to Object Methods from Interface



As shown in a previous slide, the reference used to access an object determines the methods that can be called on it. So in the case of the Interface reference shown in the slide, only the `getReturn()` method can be called. If a method receives a Returnable reference, however, and needs access to methods on Clothing or methods on Trousers, the reference can be cast to the appropriate reference type.

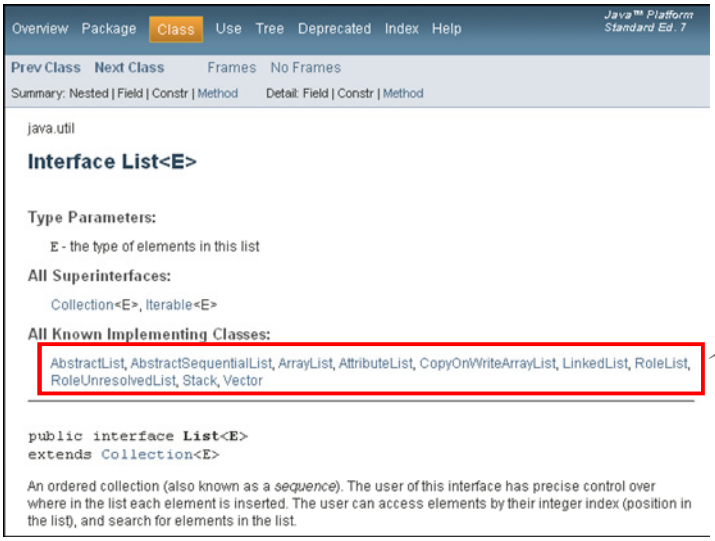
ArrayList



Some of the best examples of inheritance and the utility of Interface and Abstract types can be found in the java API. For example the ArrayList class extends AbstractList class, which itself extends AbstractCollection. AbstractCollection implements the List interface, which means that ArrayList also implements the List interface.

To use the ArrayList as a List, use the List interface as the reference type.

List Interface

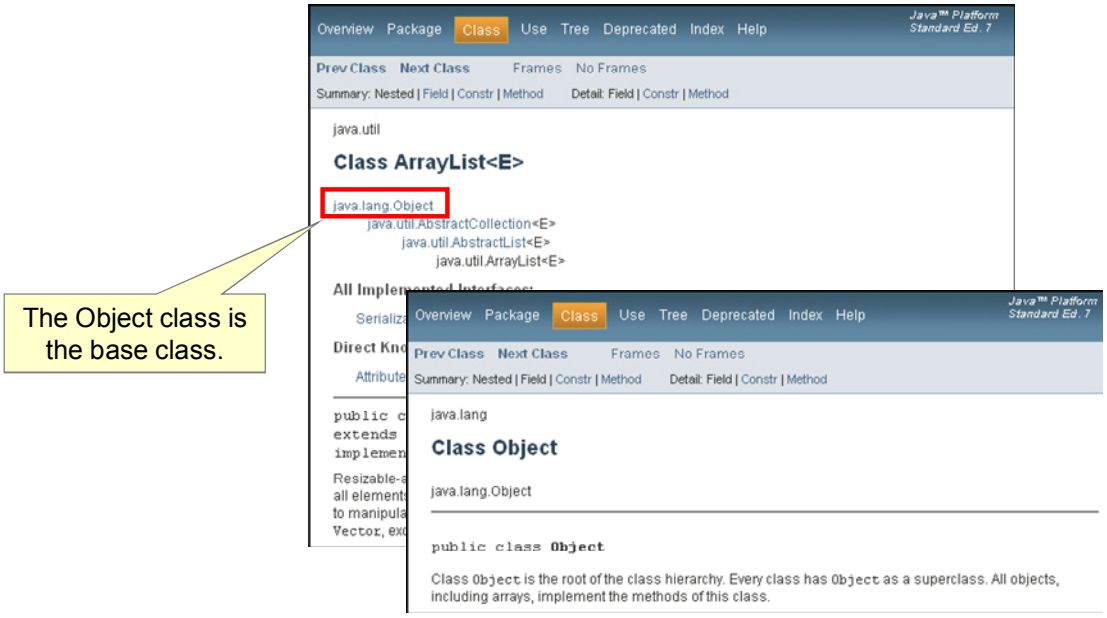


The List interface is implemented by many classes. This means that any method that requires a List may actually be passed a List reference to any objects of these types (but not the abstract classes because they cannot be instantiated).

Topics

- Overview of inheritance
- Working with superclasses and subclasses
- Polymorphism and overriding methods
- Interfaces
- The Object object

The Object Class

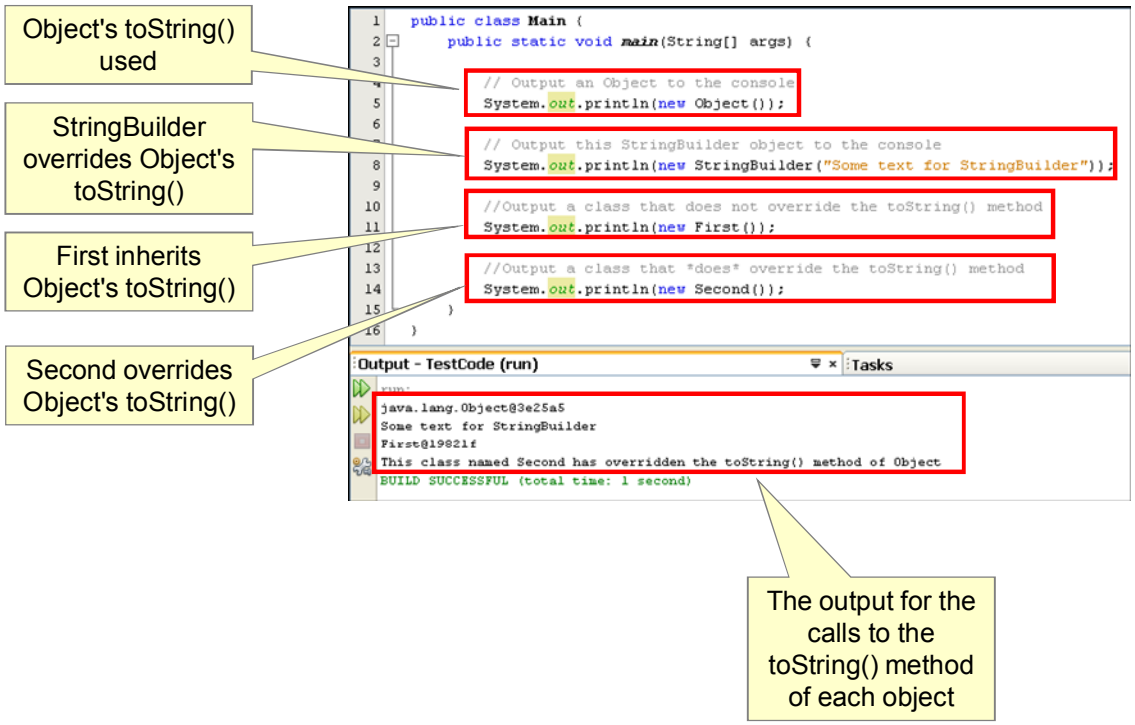


ORACLE

All classes have at the very top of their hierarchy the `Object` class. It is so central to how Java works that all classes that do not explicitly extend another class automatically extend `Object`. So all classes have `Object` at the root of their hierarchy. This means that all classes have access to the methods of `Object`. Being the root of the object hierarchy, `Object` does not have many methods—only very basic ones that all objects must have.

An interesting method is the `toString()` method. The `Object toString()` method gives very basic information about the object; generally classes will override the `toString()` method to provide more useful output. The `System.out.println()` uses the `toString()` method on an object passed to it to output a string representation.

Calling the `toString()` Method



All objects have a `toString()` method because it exists in the Object class. But the `toString()` method may return quite different results depending on whether or not that method has been overridden. In the example in the slide, `toString()` is called (via the `println()` method of `System.out`) on four objects:

- **An Object object:** This calls the `toString()` method of the base class. It returns the name of the class (`java.lang.Object`), a `@` symbol, and a hash of the object (a unique number associated with the object.
- **A StringBuilder object:** This calls the `toString()` method on the StringBuilder object. StringBuilder overrides the `toString()` method that it inherits from Object to return a String object of the set of characters it is representing.
- **An object of type First, a test class:** First is a class with no code, so the `toString()` method called is the one that is inherited from the Object class.
- **An object of type Second, a test class:** Second is a class with one method named `toString()`, so this overridden method will be the one that is called.

There is a case for re-implementing the `getDescription()` method used by the Clothing classes to instead use an overridden `toString()` method.

Quiz

Which methods of an object can be accessed via an interface that it implements?

a. All the methods implemented in the object's class
b. All the methods implemented in the object's superclass
c. The methods declared in the interface

Answer: c

Summary

In this lesson, you should have learned the following:

- Creating class hierarchies with subclasses and superclasses helps to create extensible and maintainable code by:
  - Generalizing and abstracting code that may otherwise be duplicated
  - Utilizing polymorphism
- Creating interfaces:
  - Allows you to link classes in different object hierarchies by their common behavior
  - Use an Interface reference type in your code so that the implementing class can be changed more easily.

ORACLE

Inheritance enables programmers to put common members (variables and methods) in one class and have other classes *inherit these common* members from this new class.

The class containing members common to several other classes is called the superclass or the parent class. The classes that inherit from, or extend, the superclass are called subclasses or child classes.

Inheritance also allows object methods and fields to be referred to by a reference that is either the type of the object, the type of any of its superclasses, or an interface that it implements.

Finally, inheritance enables polymorphism.

Practice 12-1: Creating and Using Superclasses and Subclasses

This practice covers the following topics:

- Designing the class hierarchy
- Creating and testing the classes

In this practice, you design and then create a class hierarchy that will form the basis for an Employee Tracking System of the Marketing department in the Duke's Choice Company. This practice comprises two sections. In the first section, you create a simple design model for the class hierarchy. In the second section, you create the actual classes and test them.

Practice 12-2: Using a Java Interface

This practice covers the following topics:

- Creating a Java interface called Printable
- Implementing it within the Employee class hierarchy

ORACLE

In this practice, you create an interface called Printable and implement it within the class hierarchy that you built in Practice 11-1. You also examine and run another small application that uses the same Printable interface to better understand the benefits of using interfaces.

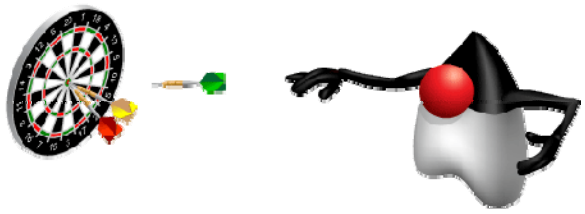
13

Handling Errors

Objectives

After completing this lesson, you should be able to:

- Describe the different kinds of errors that can occur and how they are handled in Java
- Describe what Exceptions are used for in Java
- Determine what Exceptions are thrown, for any foundation class
- Write code to handle an Exception thrown by the method of a foundation class



ORACLE

Topics

- Handling errors: an overview
- Propagation of Exceptions
- Catching and throwing Exceptions
- Multiple Exceptions and Errors

Reporting Exceptions

Coding mistake:

```
int[] intArray = new int[5];
intArray[5] = 27;
```

Output in console:

```
Exception in thread "main"
   java.lang.ArrayIndexOutOfBoundsException: 5
        at TestErrors.main(TestErrors.java:17)
```


You may have come across this while working on some of the previous practice activities. The code shows a common mistake made when accessing an array. Remember that arrays are zero based (the first element is accessed by a zero index), so in an array like the one in the slide that has five elements, the last element is actually intArray[4].

intArray[5] tries to access an element that doesn't exist and Java responds to this programming mistake by printing the text shown in the console.

Reporting Exceptions

Calling code in main():

```
TestArray myTestArray = new TestArray(5);
myTestArray.addElement(5, 23);
```

TestArray class:

```
public class TestArray {
   int[] intArray;
   public TestArray (int size) {
      intArray = new int[size];
   }
   public void addElement(int index, int value) {
     intArray[index] = value;
   }
}
```

ORACLE

Here is a very similar example, except this time the code that creates the array and tries to assign a value to a non-existent element has been moved to a different class. Notice how the error message in the console is almost identical to the previous example, but this time methods main() in TestException and addElement() in TestArray are listed.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at TestArray.addElement(TestArray.java:19)
    at TestException.main(TestException.java:20)
Java Result: 1
```

In this lesson, you learn why that message is printed to the console. You also learn how you can catch or trap the message so that it doesn't get printed to the console, and what other kinds of errors are reported by Java.

How Exceptions Are Thrown

Normal program execution:

1. Caller method calls worker method.
2. Worker method does work.
3. Worker method completes work and then execution returns to caller method.

When an Exception occurs, this sequence changes:

- Exception is thrown and either:
  - A special Exception object is passed to a special method-like catch block in the current method

    or
  - Execution returns to the caller method

Types of Exceptions

Three main types of Throwable:

- Error
  - Typically unrecoverable external error
  - Unchecked
- RuntimeException
  - Typically programming mistake
  - Unchecked
- Exception
  - Recoverable error
  - Checked (must be caught or thrown)

As mentioned in the previous slide, when an Exception is thrown, that Exception is an object that can be passed to a catch block. There are three main types of objects that can be thrown in this way, and all are derived from the class Throwable.

OutOfMemoryError

Programming mistake:

```
ArrayList theList = new ArrayList();
while(true) {
   String theString = "A test String";
   theList.add(theString);
   if (theList.size()% 1000000 == 0) {
      System.out.println("List now has " +
            theList.size()/100000 + " million elements!");
   }
}
```

Output in console:

```
List now has 240 million elements!
List now has 250 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java
    heap space
```

ORACLE

OutOfMemoryError is an Error. Throwables of type Error are typically used for exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.

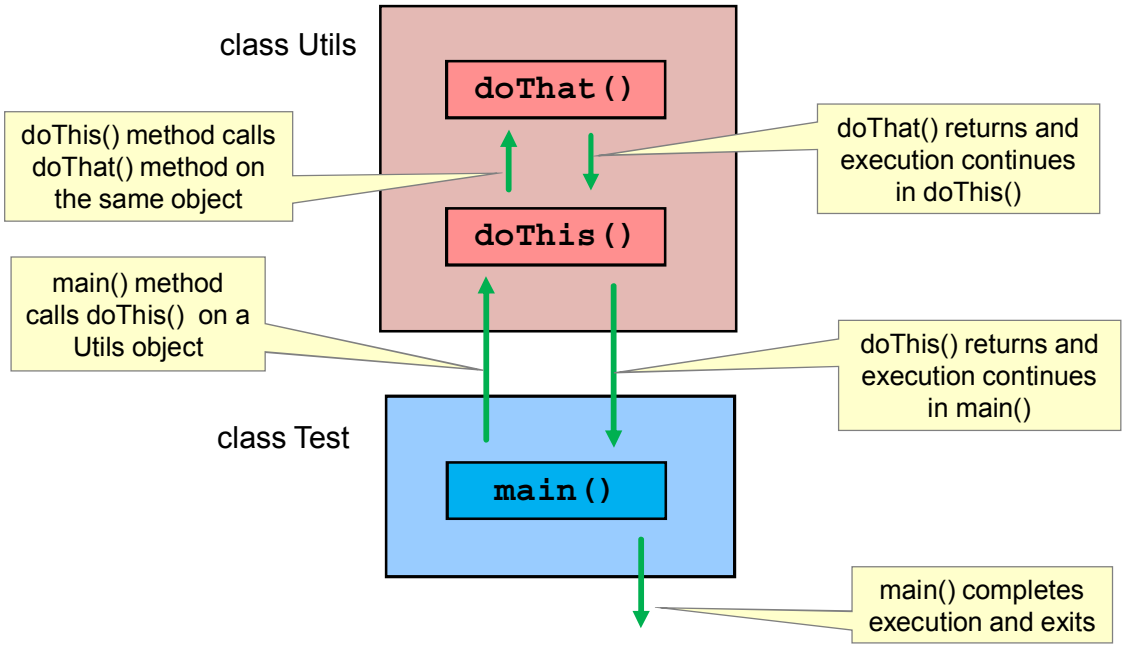
The example shown here has an infinite loop that continually adds an element to an ArrayList, guaranteeing that the JVM will run out of memory. The Error is thrown up the call stack, and since it is not caught anywhere, it is displayed in the console as shown below.

```
List now has 240 million elements!
List now has 250 million elements!
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at java.util.Arrays.copyOf(Arrays.java:2760)
        at java.util.Arrays.copyOf(Arrays.java:2734)
        at java.util.ArrayList.ensureCapacity(ArrayList.java:167)
        at java.util.ArrayList.add(ArrayList.java:351)
        at TestErrors.main(TestErrors.java:22)
```

Topics

- Handling errors an overview
- Propagation of Exceptions
- Catching and throwing Exceptions
- Multiple Exceptions and Errors

Method Stack



ORACLE

To understand Exceptions you need to think about how methods call other methods and how this can be nested deeply. The normal mode of operation is that a caller method calls a worker method, which in turn becomes a caller method and calls another worker method and so on. This sequence of methods is called the call stack.

The example shown in the slide illustrates three methods in this relationship. The main method in the class Test (a static method), instantiates an object of type Utils and calls the method doThis() on that object. The doThis() method in turn calls a private method doThat() on the same object. When it comes to the end of its code or a return statement, each method returns execution to the method that called it.

Note that as far as how methods call and return and as far as how Exceptions are thrown, the fact that there is one class method here and two instance methods on the same object is immaterial.

Call Stack: Example

Test class

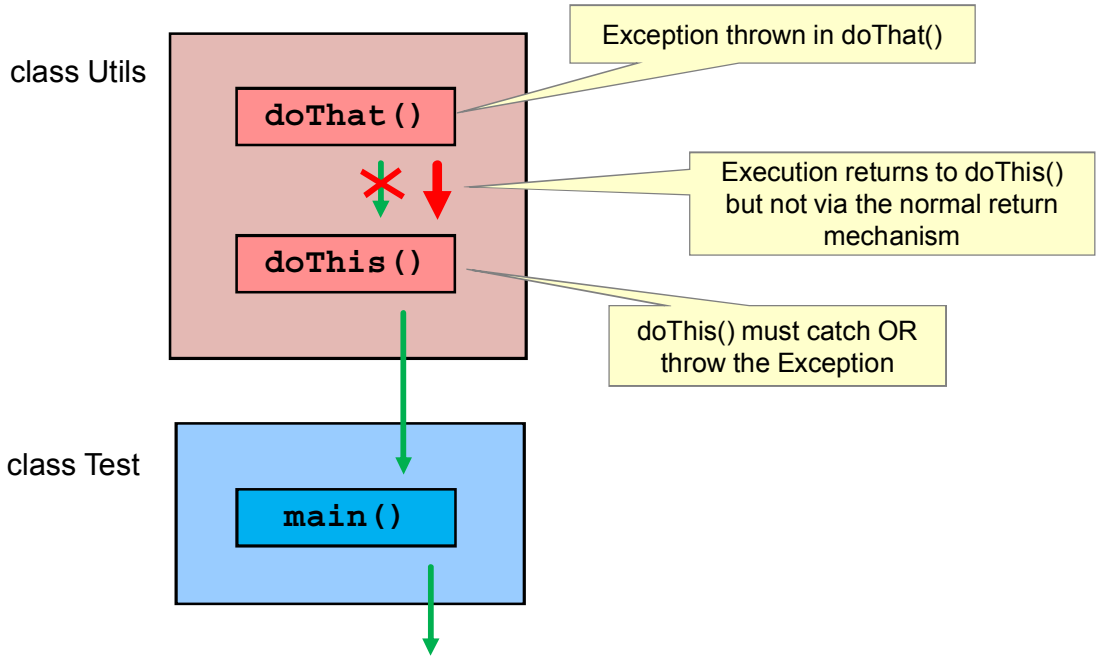
```
public static void main (String args[]) {
   Utils theUtils = new Utils();
   theUtils.doThis();
}
```

Utils class

```
  public void doThis() {
     ...< code to do something >...
     doThat();
  return;

  public void doThat() throws Exception{
     ...< code to do something >...
     if (some_problem) throw new Exception();
  return;
```


The code shown in this slide is possible code for the example illustrated in the previous slide.

Throwing Throwables

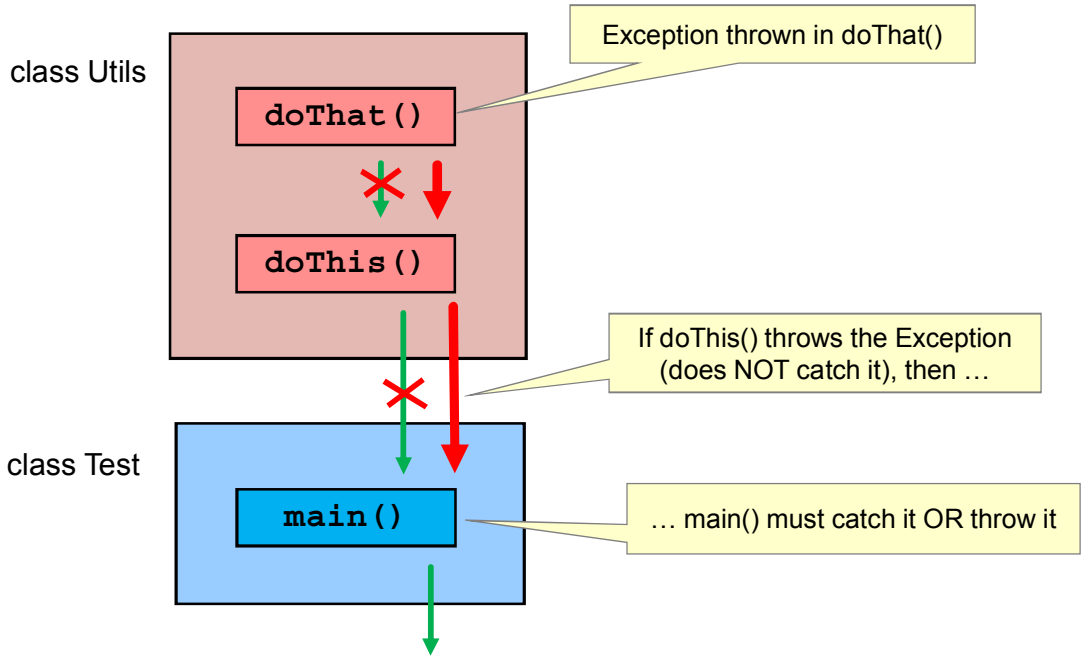


When a method finishes executing the normal flow (on completion of the method or on a return statement) goes back to the calling method and continues execution at the next line of the calling method.

When an Exception is thrown, program flow returns to the calling method, but not to the point just after the method call. Instead, if there is a try/catch block, it is thrown back to the catch block that is associated with the try block that contains the method call. If there is no try/catch block in the calling method, the Exception will be thrown back to its calling method.

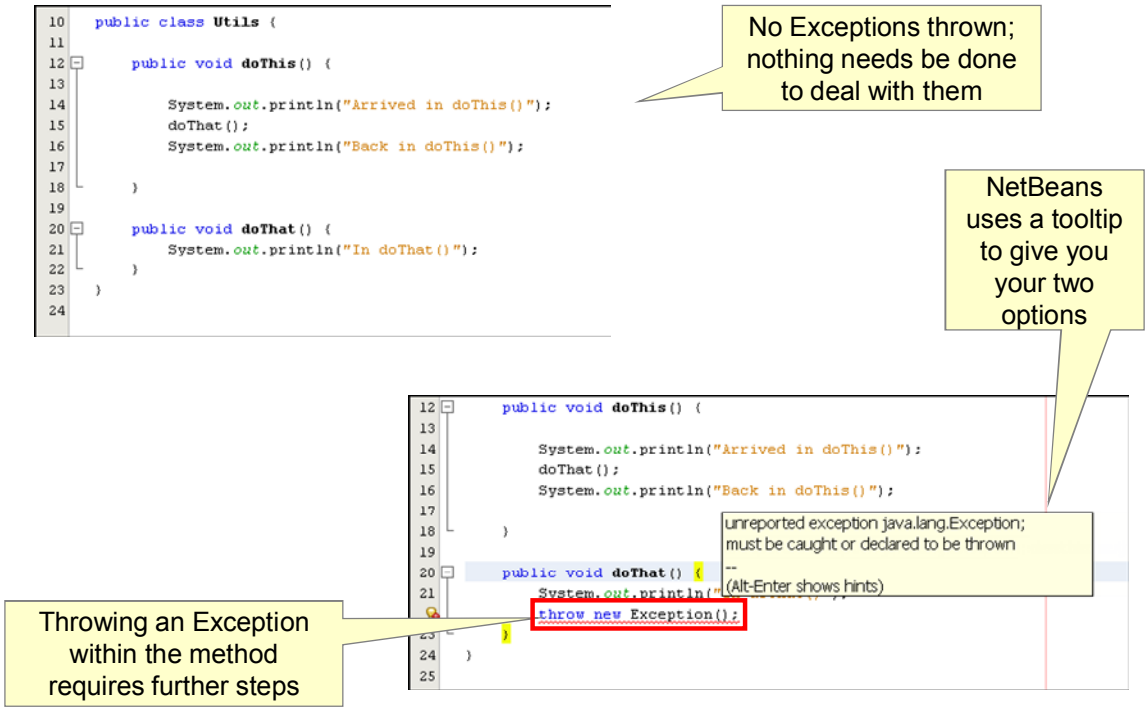
In the case of a checked Exception, this will happen because the programmer will have been forced to explicitly throw the Exception if they chose not to catch it. In the case of an Exception that is a RuntimeException or an Error, the throwing of the Exception happens automatically where no try/catch exists.

Throwing Throwables



This diagram illustrates an Exception originally thrown in doThat(), being thrown to doThis(). The error is not caught there, so it is thrown to its caller method, which is the main method.

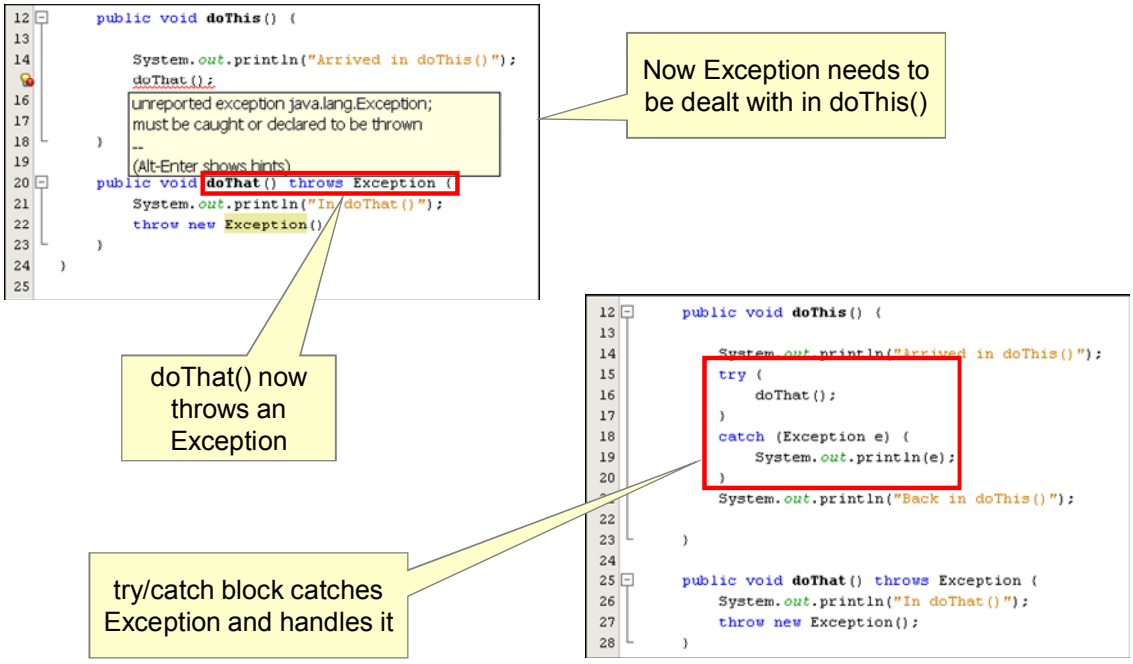
Working with Exceptions in NetBeans



Here you can see the code for the Utils class shown in NetBeans. In the first screenshot, no Exceptions are thrown, so NetBeans shows no syntax or compilation errors. In the second screenshot, doThat() throws an Exception, and NetBeans flags this as something that needs to be dealt with by the programmer. As you can see from the tooltip, it gives the two options that a programmer must choose from if handling checked Exceptions.

In these early examples, for simplicity we use the superclass, Exception. However, as you will see later, you should not throw so general an exception. Where possible, when you catch an Exception, you should try to catch a specific Exception.

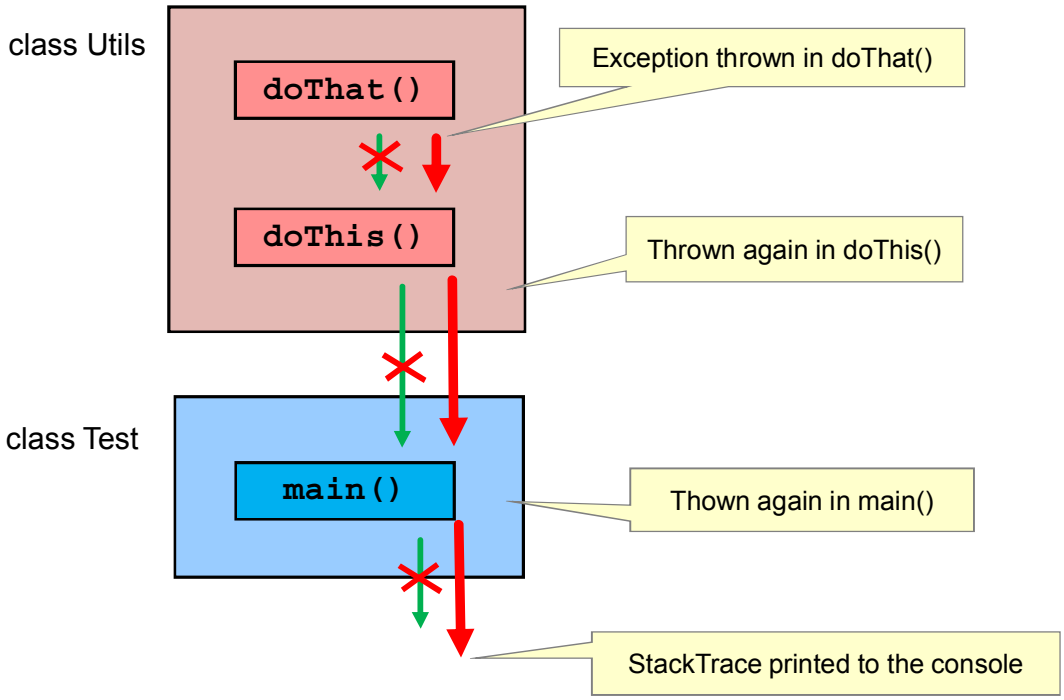
Catching an Exception



Here you can see that the Exception thrown in `doThat()` has been handled by:

- Adding `throws Exception` to the `doThat()` method signature, ensuring that it will be thrown to the caller, doThat()
- Adding a try/catch block to `doThis()` so that:
  - The try block contains the call to `doThat()`
  - The catch block is set up with the parameter `Exception`

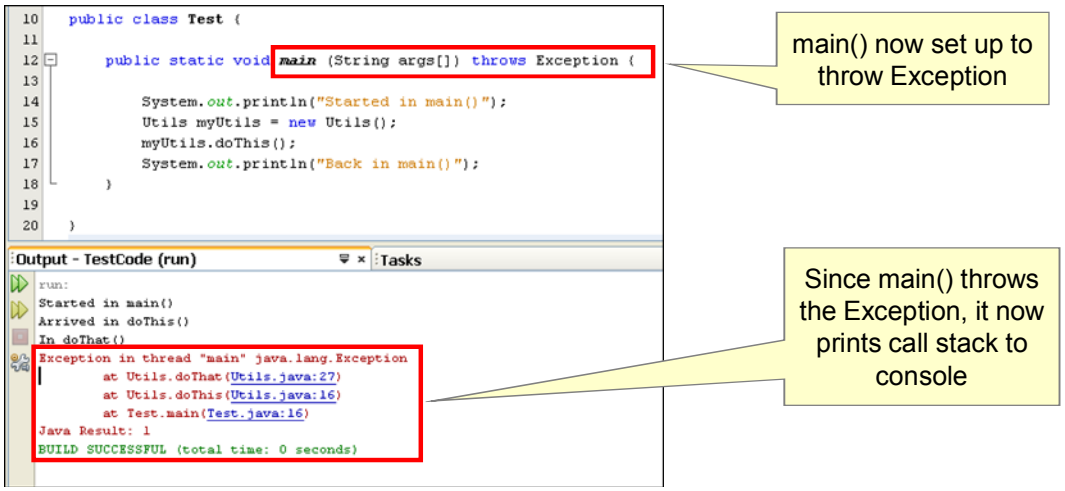
Uncaught Exception



But what happens if none of the methods in the call stack have try/catch blocks? That situation is illustrated by the diagram shown in this slide. Since there are no try/catch blocks, the Exception is thrown all the way up the call stack. But what does it mean to throw an Exception from the main() method? This will cause the program to exit, and the Exception, plus a stack trace for the Exception, will be printed to the console.

Exception Printed to Console

Example of main() throwing exception



In the example, you can see what happens when the Exception is thrown up the call stack all the way to the main() method, and it throws the Exception too.

Did you notice how similar this looks to the first example you saw of an ArrayIndexOutOfBoundsException? In both cases, the Exception is displayed as a stack trace to the console.

However, there was something different about the ArrayIndexOutOfBoundsException: None of the methods threw that Exception! So how did it get passed up the call stack?

The answer is that ArrayIndexOutOfBoundsException is a RuntimeException, and RunTimeExceptions are different from Exceptions. They are automatically thrown up the call stack without this being explicitly declared in the method signature.

Summary of Exception Types

A Throwable is a special type of Java object:

- Only object type used as the argument in a catch clause
- Only object type that can be "thrown" to the calling method
- Has two subclasses:
  - Error
    - Automatically thrown to the calling method if created
  - Exception
    - Must be explicitly thrown to the calling method

    OR
    - Caught using a try catch block
    - Has a subclass RuntimeException which is automatically thrown to the calling method

ORACLE

Exceptions that are not also RuntimeExceptions must be explicitly handled. The next examples show how you would work with an IOException.

Quiz

Which one of the following statements is true?

a. A RuntimeException must be caught.
b. A RuntimeException must be thrown.
c. A RuntimeException must be caught or thrown.
d. A RuntimeException is thrown automatically.

Answer: d

Quiz

Which of the following objects are checked Exceptions?

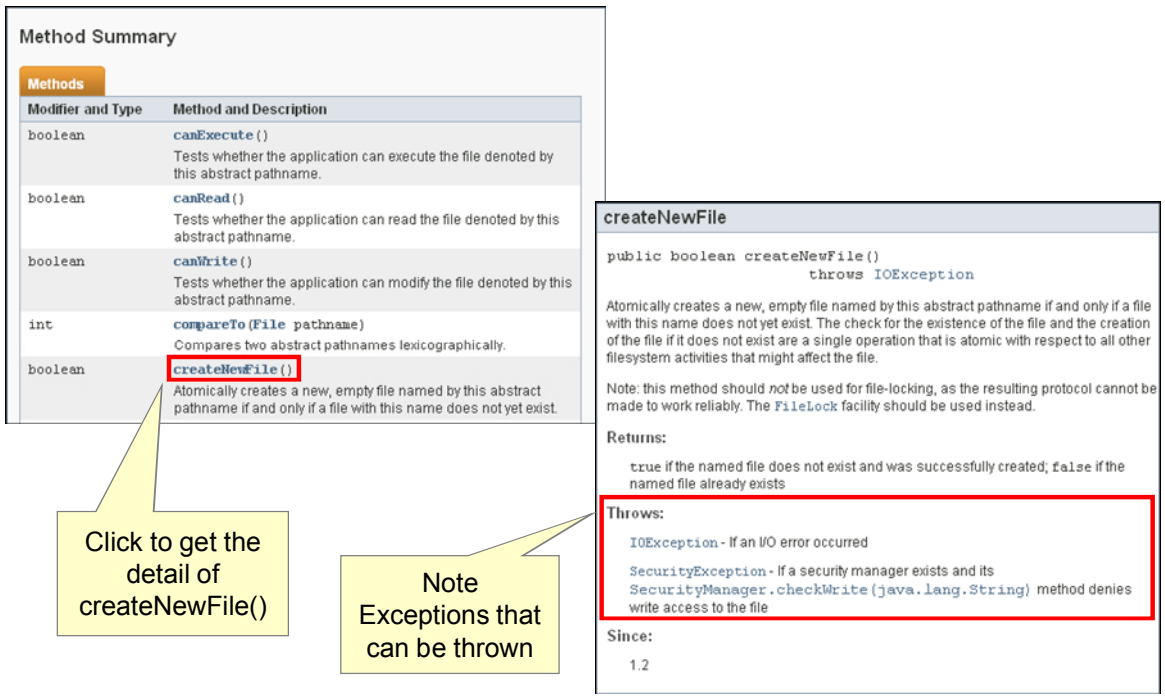
a. All objects of type Throwable
b. All objects of type Exception
c. All objects of type Exception that are not of type RuntimeException
d. All objects of type Error
e. All objects of type RuntimeException

Answer: c

Topics

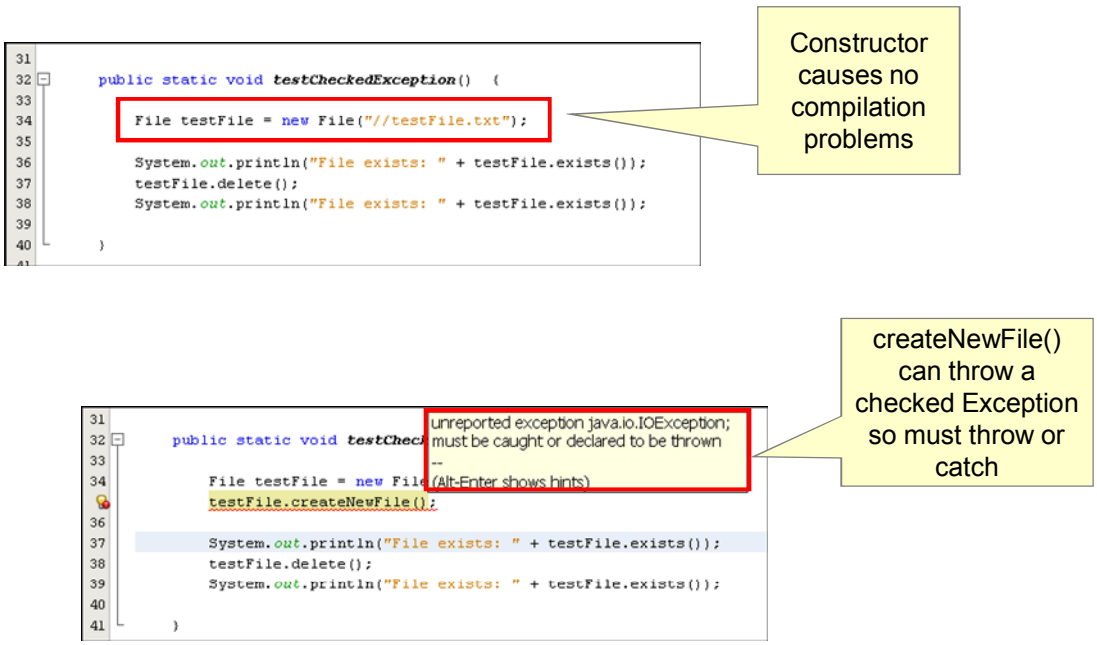
- Handling errors: an overview
- Propagation of Exceptions
- Catching and throwing Exceptions
- Multiple Exceptions and Errors

Exceptions in the Java API Documentation



It is necessary when working with any API to determine what Exceptions are thrown by the object's constructors or methods. The example in the slide is for the File class. File has a method createNewFile() that can throw an IOException or a SecurityException. SecurityException is a RuntimeException so SecurityException is unchecked, but IOException is a checked Exception.

Calling a Method That Throws an Exception



ORACLE

The two screenshots in the slide show a simple method testCheckedException(). In the first example the File object is created using the constructor. Note that even though the constructor can throw a NullPointerException (if the constructor argument is null), you are not forced to catch this Exception.

However, in the second example, createNewFile() can throw an IOException, and NetBeans shows that you must deal with this.

Working with a Checked Exception

Catching IOException:

```
public static void main(String args[]) {
   try {
      testCheckedException();
   }
   catch (IOException e) {
      System.out.println(e);
   }
 }

public static void testCheckedException() throws IOException{
   File testFile = new File("//testFile.txt");
   testFile.createNewFile();
   System.out.println("File exists: " + testFile.exists());

}
```

ORACLE

The example is handling the possible raised Exception by:

- Throwing the Exception from the testCheckedException() method
- Catching the Exception in the caller method

In this example the catch method will catch the Exception because the path to the text file is not correctly formatted. The System.out.println(e) will call the toString() method of the Exception and the result will be:

```
java.io.IOException: The filename, directory name, or volume label
syntax is incorrect
```

Best Practices

- Catch the actual Exception thrown, not the Exception or Throwable superclass.
- Examine the Exception to find out the exact problem so you can recover cleanly.
- You don't need to catch every Exception.
  - A programming mistake should not get handled. It must be fixed.
  - Ask yourself, "Does this Exception represent behavior I want the program to recover from?"

Bad Practices

```
public static void main (String args[]) {
  try {
     createFile("c://testFile.txt");
  }
  catch (Exception e) {

     System.out.println("Problem creating the file!");
     ...< other actions >...
  }
}

public static void createFile(String fileName) throws
   IOException {
     File f = new File(fileName);
     f.createNewFile();

     int[] intArray = new int[5];
     intArray[5] = 27;
  }
```

Catching superclass?

No processing of the Exception object?

ORACLE

The code in the slide illustrates two poor programming practices.

1. The catch clause catches an Exception rather than the expected Exception from calling the createFile method (IOException).
2. The catch clause doesn't analyze the Exception object and instead simply assumes that the expected Exception has been thrown from the File object.

A major drawback of this careless programming style is shown by the fact that the code will print the following message to the console.

```
There is a problem creating the file!
```

This suggests that the file has not been created, and indeed any further code in the catch block will run. But what is actually happening in the code?

Bad Practices

```
public static void main (String args[]) {
  try {
     createFile("c://testFile.txt");
  }
  catch (Exception e) {
       System.out.println(e);
     ...< other actions >...
  }
}

public static void createFile(String fileName) throws
   IOException {
     File f = new File(fileName);
     System.out.println(fileName + " exists? " + f.exists());
     f.createNewFile();
     System.out.println(fileName + " exists? " + f.exists());
     int[] intArray = new int[5];
     intArray[5] = 27;
  }
```

What is the object type?

toString() will be called on this object

ORACLE

Putting in a few System.out.println() calls in the createFile method may help clarify what is happening. The output now will be:

```
C://testFile.txt exists? false
C://testFile.txt exists? true
java.lang.ArrayIndexOutOfBoundsException: 5
```

So the file is being created! And we can see that the Exception is actually an ArrayIndexOutOfBoundsException that is being thrown by the final line of code in createFile().

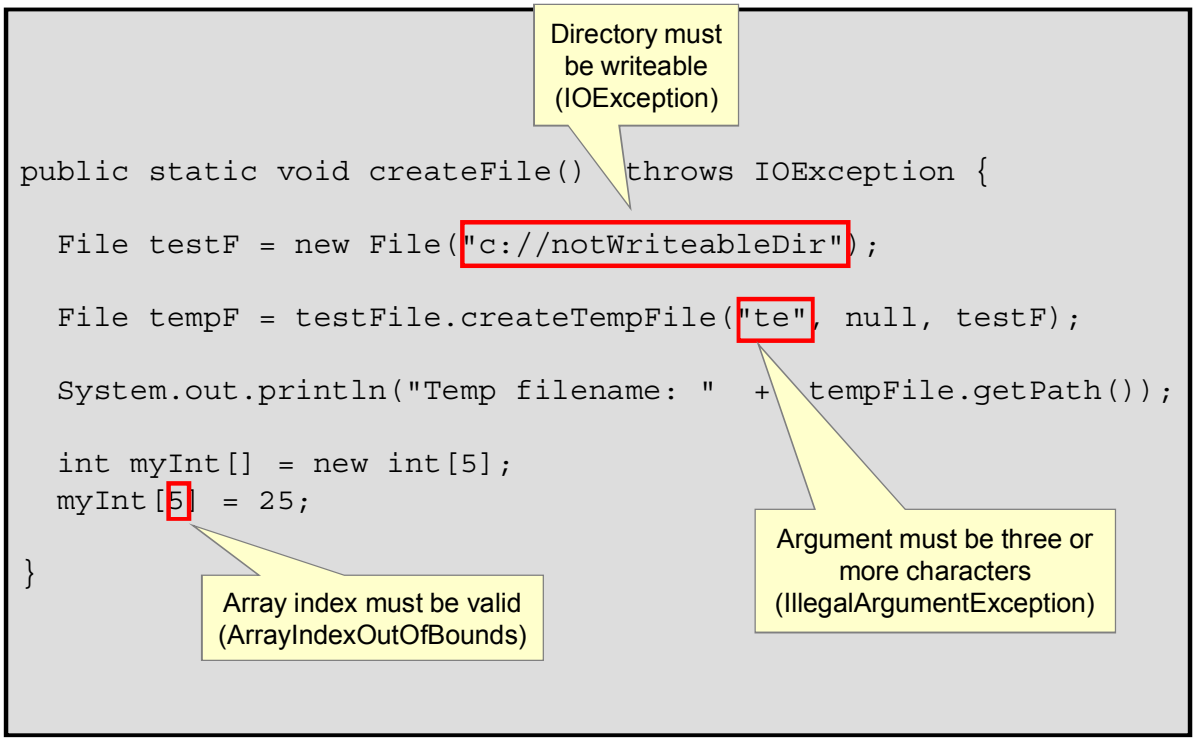
In this example it is obvious that the array assignment can throw an Exception, but it may not be so obvious. In this case, the `createNewFile()` method of File actually throws another Exception—a SecurityException. Since it is an unchecked Exception, it will be thrown automatically.

If you check for the specific Exception in the catch clause, you will remove the danger of assuming what the problem is.

Topics

- Handling errors: an overview
- Propagation of Exceptions
- Catching and throwing Exceptions
- Multiple Exceptions and Errors

Multiple Exceptions



The example shows a method that could potentially throw three different Exceptions. It uses the File method createTempFile(), which creates a temporary file (it ensures that each call will create a new and different file and also can be set up so that the temporary files created will be deleted on exit).

The three different Exceptions are the following:

`IOException`

C:\notWriteableDir is a directory but it is not writeable. This will cause createTempFile() to throw an IOException (checked).

`IllegalArgumentException`

The first argument passed to createTempFile should be three or more characters long. If it is not, the method will throw an IllegalArgumentException (unchecked).

`ArrayIndexOutOfBoundsException`

As in previous examples, trying to access a non-existent index of an array throws an `ArrayIndexOutOfBoundsException` (unchecked).

Catching IOException

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   }
}

public static void createFile()  throws IOException {

  File testF = new File("c://notWriteableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

ORACLE

The example shows the minimum Exception handling (the compiler will insist on at least the IOException being handled).

With the directory set as shown at c://notWriteableDir, the output of this code will be:

```
java.io.IOException: Permission denied
```

However, if the file is set as c://writeableDir (a writeable directory), the output will now be:

```
Exception in thread "main" java.lang.IllegalArgumentException: Prefix
string too short
   at java.io.File.createTempFile(File.java:1782)
   at
MultipleExceptionExample.createFile(MultipleExceptionExample.java:34)
   at MultipleExceptionExample.main(MultipleExceptionExample.java:18)
```

The argument, "te", is causing an IllegalArgumentException to be thrown, and since it is a RuntimeException, it gets thrown all the way out to the console.

Catching IllegalArgumentException

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   } catch (IllegalArgumentException iae) {
      System.out.println(iae);
   }
}

public static void createFile()  throws IOException {

  File testF = new File("c://writeableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```


The example shows a further catch clause added to catch the potential IllegalArgumentException

With the first argument of the createTempFile() method set to "te" (fewer than three characters), the output of this code will be:

```
java.lang.IllegalArgumentException: Prefix string too short
```

However, if the argument is set to "temp", the output will now be:

```
Temp filename is /Users/kenny/writeableDir/temp938006797831220170.tmp
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
... < some code omitted > ...
```

Now the temporary file is being created, but there is still another argument being thrown by the createFile() method. And as ArrayIndexOutOfBoundsException is a RuntimeException, it is automatically thrown all the way out to the console.

Catching Remaining Exceptions

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   } catch (IllegalArgumentException iae) {
      System.out.println(iae);
   } catch (Exception e) {
      System.out.println(e);
   }
}
public static void createFile()  throws IOException {
  File testF = new File("c://writeableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```


The example shows a further catch clause added to catch all the remaining Exceptions.

For the example code, the output of this code will be:

```
Temp filename is /Users/kenny/writeableDir/temp7999507294858924682.tmp
java.lang.ArrayIndexOutOfBoundsException: 5
```

Finally, the catch Exception clause can be added to catch any further Exceptions.

Summary

In this lesson, you should have learned how to:

- Describe the different kinds of errors that can occur and how they are handled in Java
- Describe what Exceptions are used for in Java
- Determine what Exceptions are thrown, for any foundation class
- Write code to handle an Exception thrown by the method of a foundation class

ORACLE

Practice 13-1 Overview: Using a try/catch Block to Handle an Exception

This practice covers the following topics:

- Using the Java documentation to determine if an exception is checked or unchecked
- Using the Java documentation to determine the nature of exceptions thrown by different methods of a class
- Writing a try/catch block to catch an error

ORACLE

Practice 13-2 Overview: Catching and Throwing a Custom Exception

This practice covers the following topics:

- Writing an exception handler to determine valid employee skills as part of a catch/throw block
- Using that exception handler in different classes

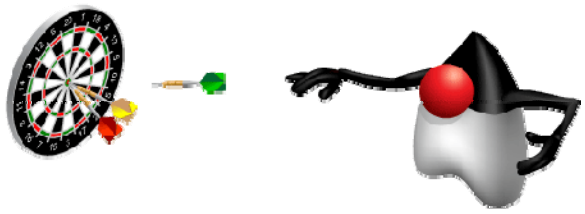
14

The Big Picture

Objectives

After completing this lesson, you should be able to:

- Describe how to deploy an application
- Describe a complete Java application that includes a database back end

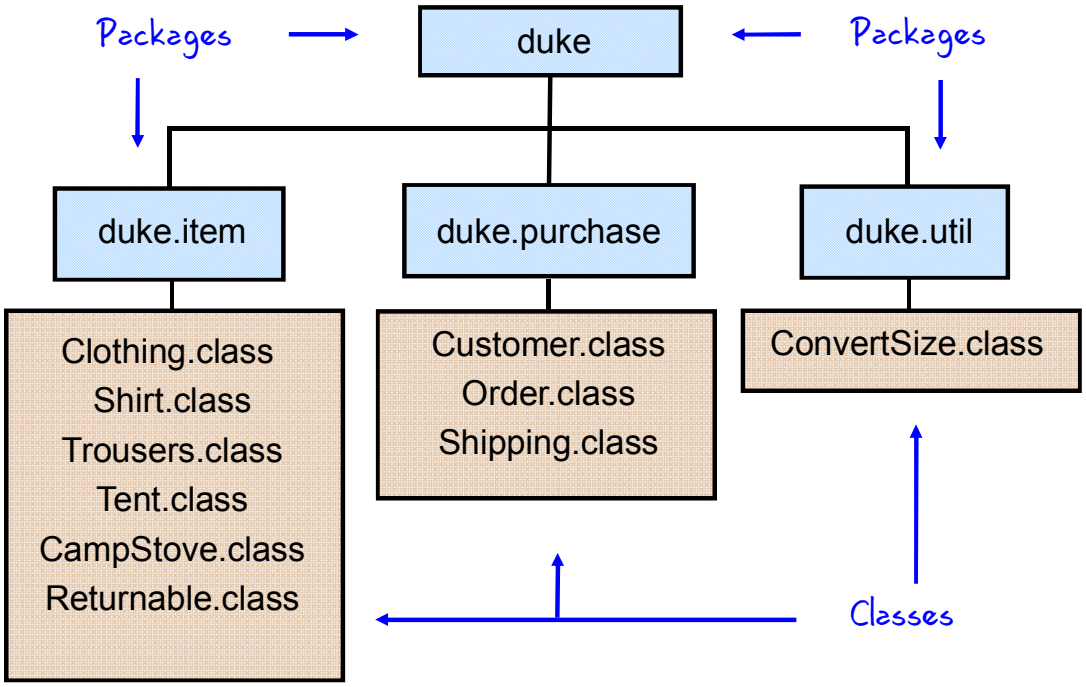


ORACLE

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Duke's Choice application
- Application modifications and enhancements

Packages

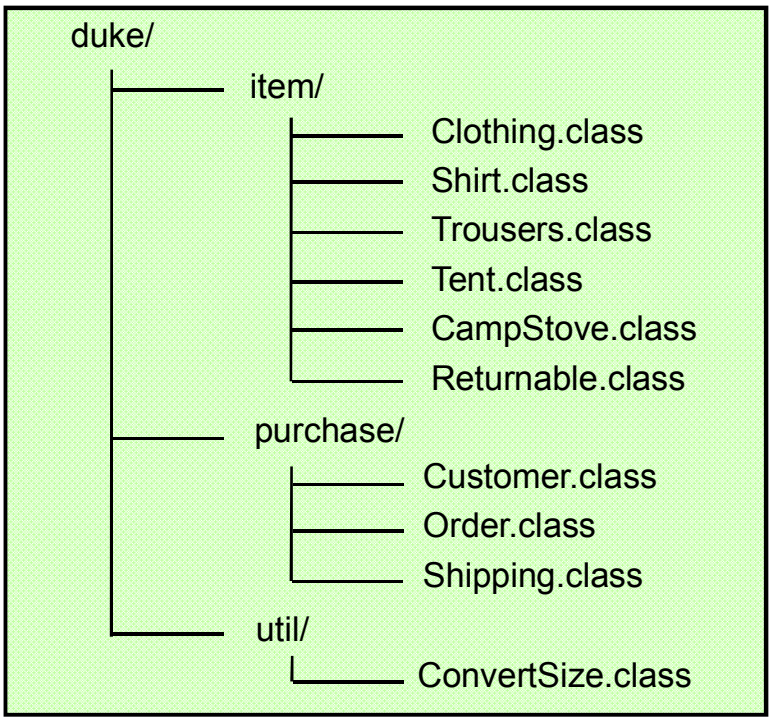


Classes are grouped into packages to ease the management of the system

There are many ways to group classes into meaningful packages. There is no right or wrong way; but a common technique is to group classes into a package by semantic similarity.

For example, the software for Duke's Choice could contain a set of item classes(such as `Shirt`, `Trousers`, `Tent`, the superclasses `Clothing` and `Camping`, and so on), a set of objects that use these item objects to arrange a purchases, and a set of utilities. All of these packages are contained in the top-level package called `duke`.

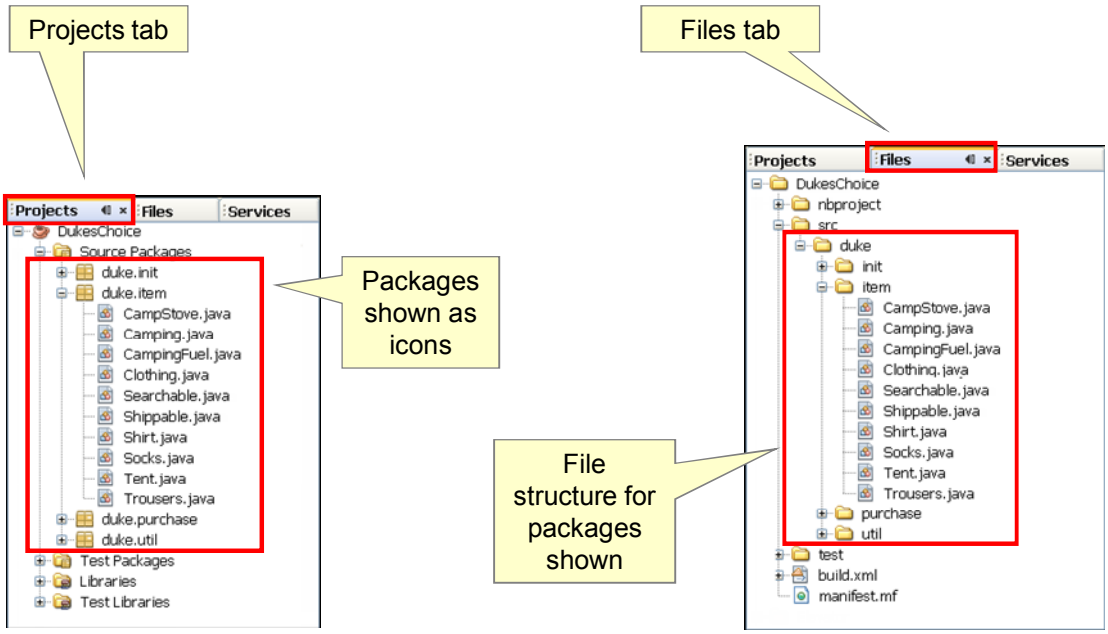
Packages Directory Structure



ORACLE

Packages are stored in a directory tree containing directories that match the package names. For example, the `Clothing.class` file should exist in the directory `item`, which is itself contained in the directory `duke`.

Packages in NetBeans



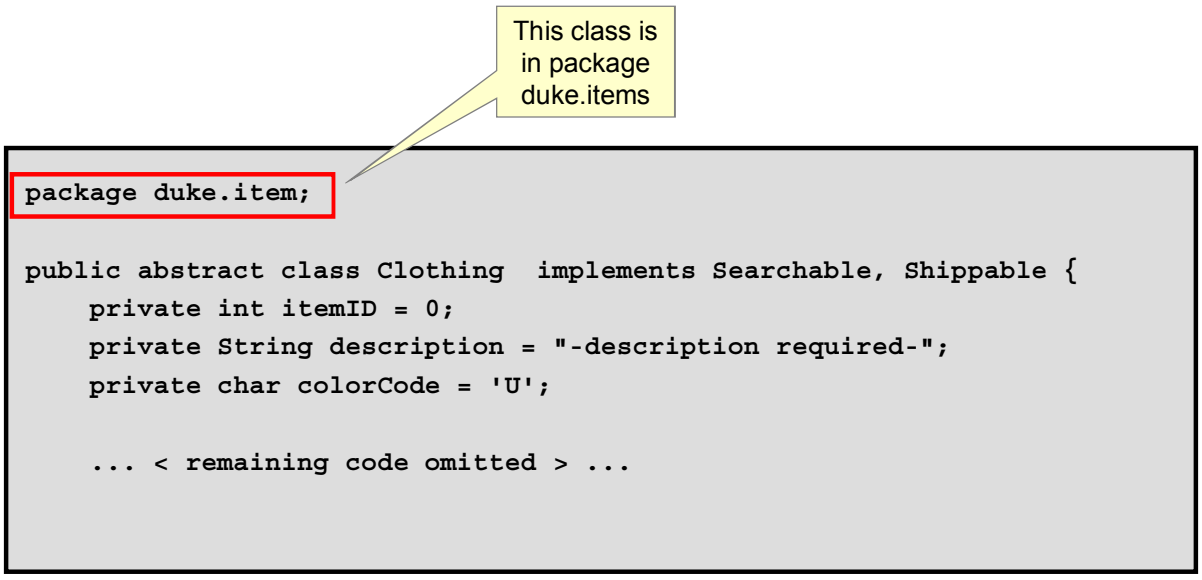
ORACLE

The left panel in NetBeans has three tabs. Two of these tabs, Projects and Files, show how packages relate to the file structure.

The Projects tab shows the packages and libraries for each project (the screenshot shows only DukesChoice). The source package shown is the one containing the packages and classes for Duke's Choice, and the screenshot shows the four packages `duke.init`, `duke.item`, `duke.purchase`, and `duke.util`. Each of these packages can be expanded to show the source files within, as has been done for the `duke.item` package in the screenshot.

The Files tab shows the directory structure for each project. In the screenshot you can see how the packages listed under the Projects tab have a corresponding directory structure. For example, the package `duke.item` has the corresponding file structure of the directory `duke` just under the directory `src`, and contains the directory `item`, which in turn contains all the source files in the package.

Packages in Source Code



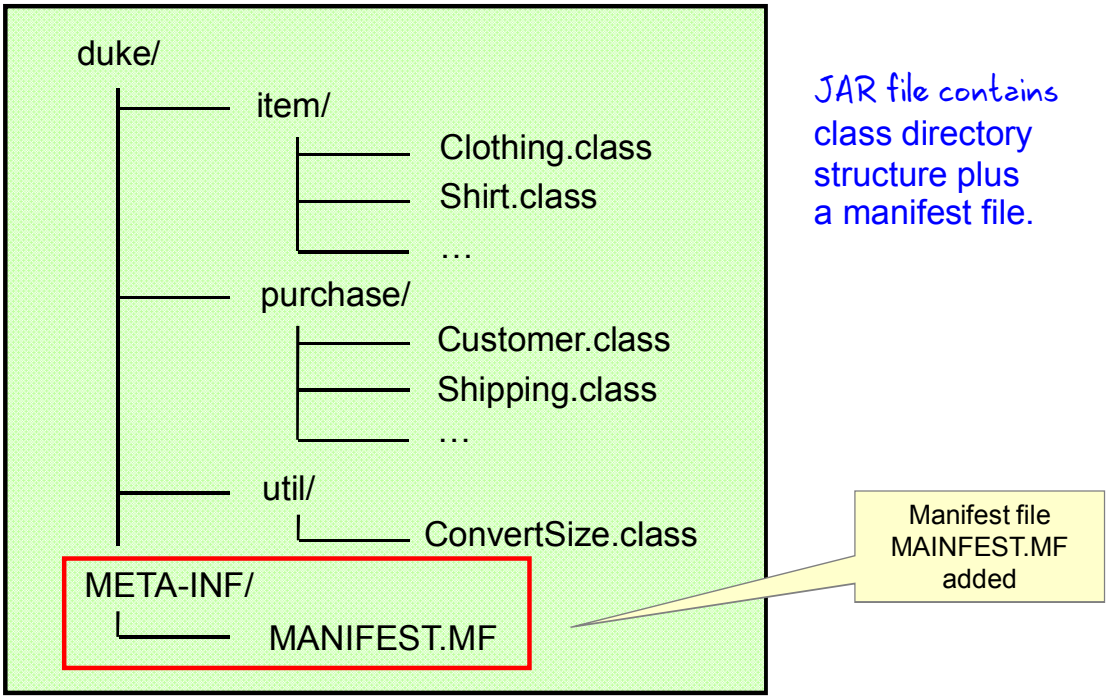
The package that a class belongs to is defined in the source code.

The example code in the slide shows the package statement being used to define the package that the Clothing class is in. Just as the class itself must be in a file of the same name as the class, the file (in this case `Clothing.java`) must be contained in a directory structure the matches the package name.

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Duke's Choice application
- Application modifications and enhancements

DukesChoice.jar

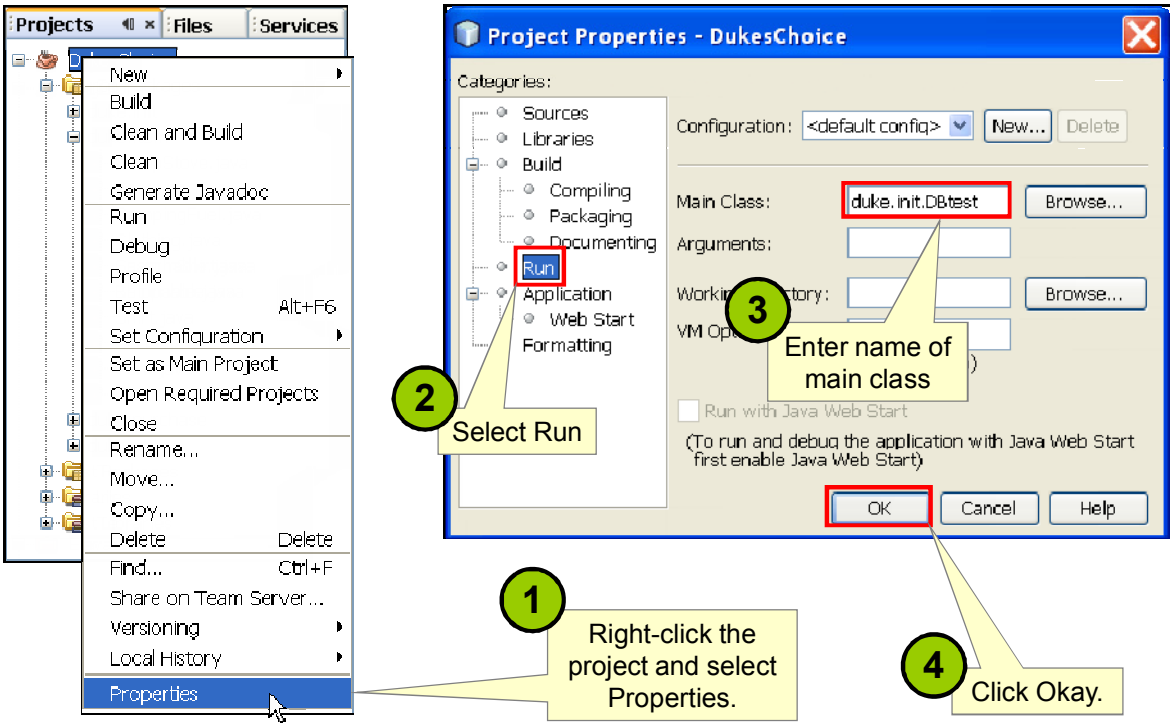


Generally, to deploy a Java application, you put the necessary files into a JAR file. This greatly simplifies running the application on another machine.

A JAR file is very like a zip file (or a tar file on UNIX) and contains the entire directory structure for the compiled classes plus an additional MANIFEST.MF file in the META-INF directory. This MANIFEST.MF file tells the Java runtime which file contains the `main()` method.

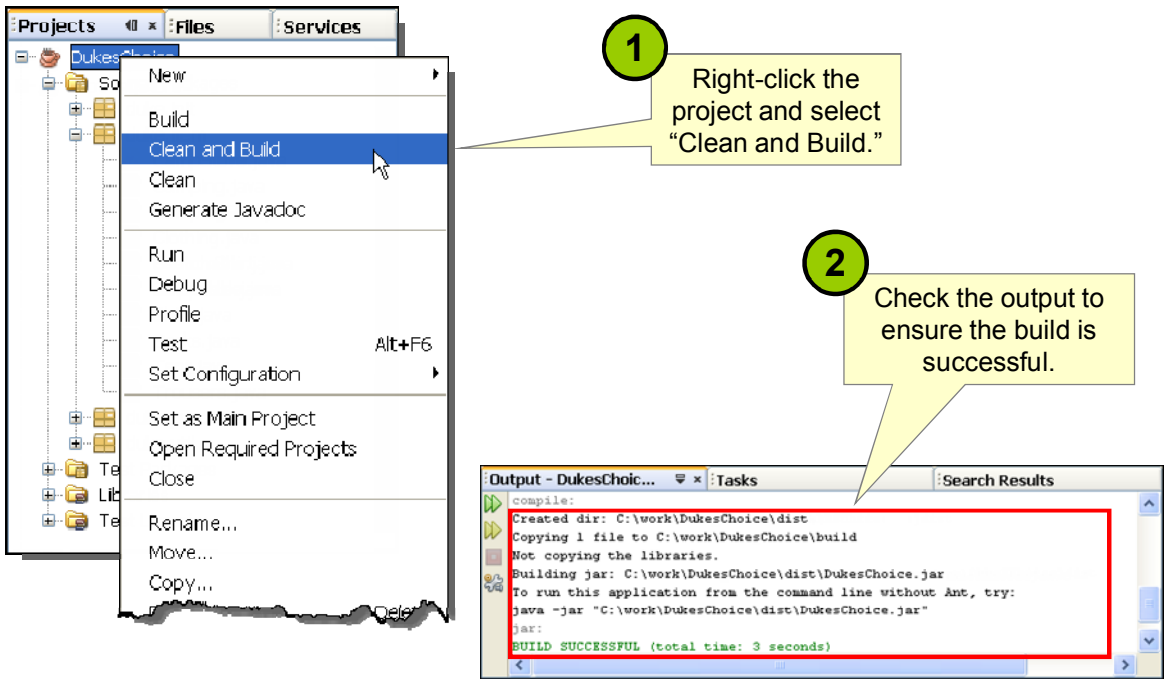
You can create a JAR file using a command-line tool called jar, but most IDEs make the creation easier, so in the next slides you'll see how to create a JAR file using NetBeans.

Set Main Class of Project



Before you create the JAR file, you need to indicate which file contains the `main()` method. This will subsequently be written to the MANIFEST.MF file.

Creating the JAR File with NetBeans



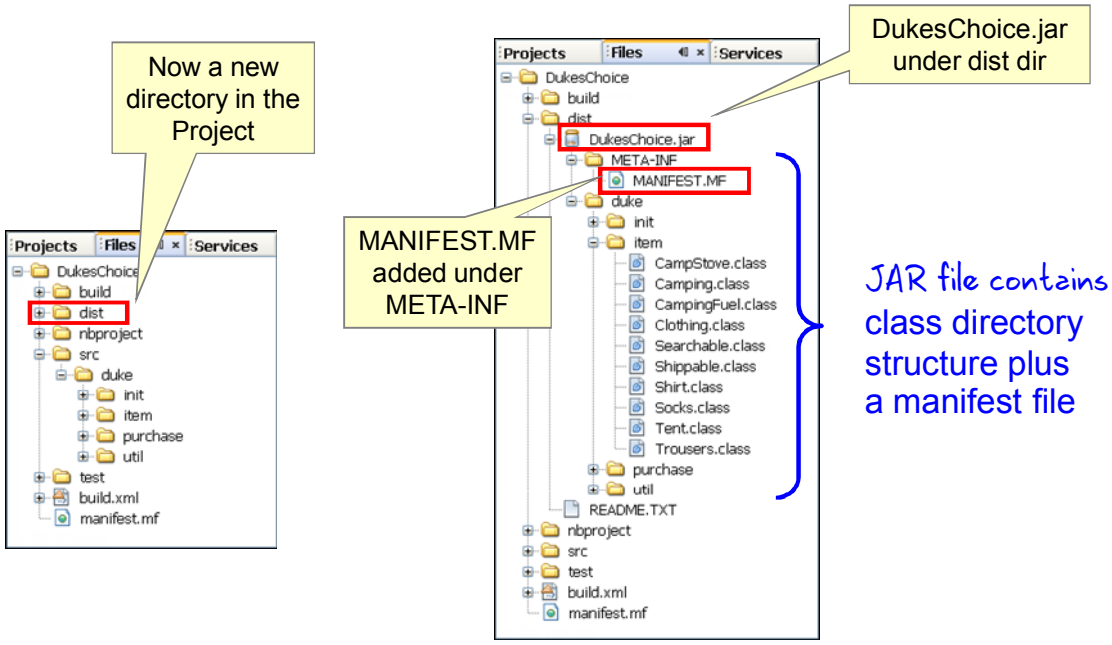
ORACLE

You create the JAR file by right-clicking the project and clicking "Clean and Build." For a small project like DukesChoice, this will take seconds.

- Clean removes any previous builds.
- Build creates a new JAR file.

You can also run "Clean" and "Build" separately.

Creating the JAR File with NetBeans



By default the JAR file will be placed in the dist directory (this directory is removed in the clean process and recreated during build). Using the files tab of NetBeans, you can look inside the JAR file and make sure that all the correct classes have been added.

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Duke's Choice application
- Application modifications and enhancements

Client/Server Two-Tier Architecture

Client/server computing involves two or more computers sharing tasks:

- Each computer is performs logic appropriate to its design and stated function.
- Front-end client communicates with back-end database.
- Client requests data from back-end.
- Server returns appropriate results.
- Client handles and displays data.

A major performance penalty is paid in two-tier client/server. The client software ends up larger and more complex because most of the logic is handled there. The use of server-side logic is limited to database operations. The client here is referred to as a *thick client.*

Thick clients tend to produce frequent network traffic for remote database access. This works well for intranet-based and local area network (LAN)-based network topologies but produces a large footprint on the desktop in terms of disk and memory requirements. Also, not all back-end database servers are the same in terms of server logic offered, and all of them have their own API sets that programmers must use to optimize and scale performance.

Client/Server Three-Tier Architecture

Three-tier client/server is a more complex, flexible approach.

- Each tier can be replaced by a different implementation:
  - Presentation could be GUI, web, smartphone, or even console.
  - Business logic defines business rules.
  - Data tier is an encapsulation of all existing data sources.

The three components or tiers of a three-tier client/server environment are *presentation, business logic or functionality,* and *data.* They are separated so that the software for any one of the tiers can be replaced by a different implementation without affecting the other tiers.

For example, if you wanted to replace a character-oriented screen (or screens) with a GUI (the presentation tier), you would write the GUI using an established API or interface to access the same functionality programs in the character-oriented screens.

The business logic offers functionality in terms of defining all of the business rules through which the data can be manipulated. Changes to business policies can affect this layer without having any impact on the actual databases.

The third tier, or data tier, includes existing systems, applications, and data that has been encapsulated to take advantage of this architecture with minimal transitional programming effort.

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- The Duke's Choice application
- Application modifications and enhancements

The Duke's Choice Application

- Abstract classes
  - Clothing
    - Extended by Shirt and other clothing classes
  - Camping
    - Extended byTent and other camping classes
- Interfaces
  - Searchable
    - All purchasable items implement Searchable.
  - Returnable
    - Items that can be returned implement Returnable.
  - Shippable
    - Items that can be shipped implement Shippable.

ORACLE

A version of the Duke's Choice application has been created to illustrate aspects of object-oriented programming in Java.

The `Clothing` Class

```
package duke.item;
public abstract class Clothing  implements Searchable, Shippable {
    private String sku = "";
    private int itemID = 0; // Default ID for all clothing items
    private String description = "-description required-"; // default
    private char colorCode = 'U'; // Exception if invalid color code?
    private double price = 0.0; // Default price for all items
    private int quantityInStock = 0;

  public Clothing(int itemID, String description, char colorCode,
                  double price, int quantityInStock ) {
    this.itemID = itemID;
    this.description = description;
    this.colorCode = colorCode;
    this.price = price;
    this.quantityInStock = quantityInStock;
    this.sku = "" + itemID + colorCode;
        ... < more code follows > ...
```

ORACLE

The Clothing class is very similar to the Shirt class you have seen earlier in the course. However, to ensure that there is a unique code for every type of item, a field SKU (Stock Keeping Unit) has been added.

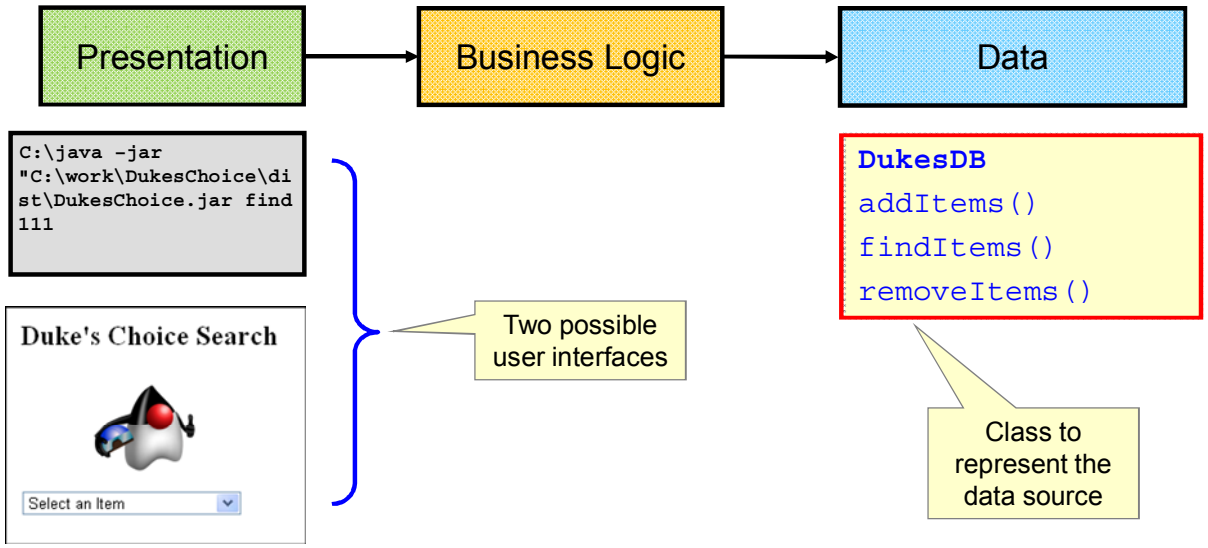
The `Clothing` Class

```
public String getDisplay(String separator) {

  String displayString = "SKU: " + getSku() + separator +
  "Item: " + description + separator +
  "Price: " + price + separator +
  "Color: " + colorCode + separator +
  "Available: " + quantityInStock;
  return displayString;
}


      ... < more code follows > ...
```

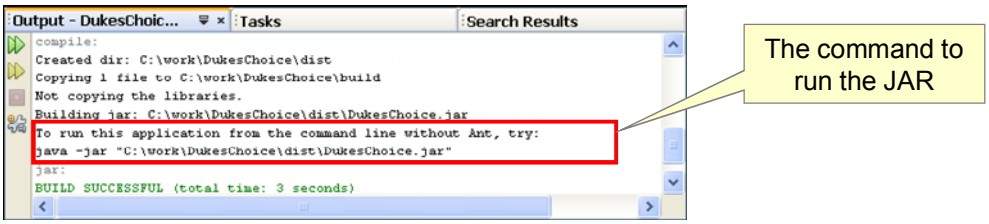
ORACLE

In addition to the previous method, display(), to display details of the item, a getDisplay() method has been added that returns a String. This allows the method to be called by different clients. It takes one argument, a String that determines how the individual attributes of the item will be separated. For example, they could be separated with a new line in the console version of the application, or with an HTML element for the web application.

The Tiers of Duke's Choice



Running the JAR File from the Command Line




```
C:\java -jar "C:\work\DukesChoice\dist\DukesChoice.jar
```

Output:

```
Please add parameters in the format:
   find <item id number>
   OR
   remove <sku> <number to remove>
```


Running the command-line application using the JAR file is very straightforward and the instructions are actually given in the output window for the build process. (If it were implemented as a GUI application it would be run the same way.)

Assuming the application is an early command-line version of the software that has been sent to Duke's Choice for testing, you would run it as shown in the slide. Because it's an early version, assume it is just for the use of Duke's Choice employees and requires parameters to be added at the command line to do anything.

Listing Items from the Command Line


```
C:\java -jar "C:\work\DukesChoice\dist\DukesChoice.jar find 111
```

Output:

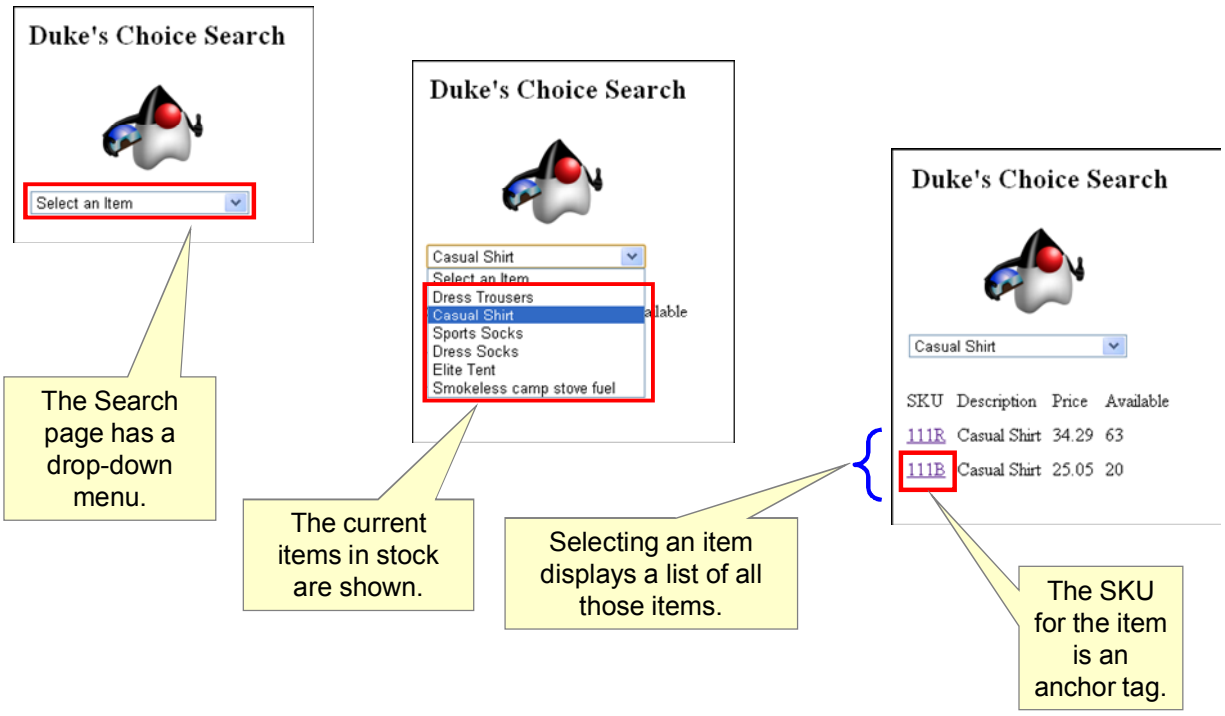
```
-------------------------------------------------------------------------
SKU: 111R | Item: Casual Shirt | Price: 34.29 | Color: R | Available: 63
-------------------------------------------------------------------------
SKU: 111B | Item: Casual Shirt | Price: 25.05 | Color: B | Available: 20
-------------------------------------------------------------------------
```

ORACLE

In this simple application, commands are entered using command-line parameters and SKU or item IDs. So you can assume that Duke's Choice employees have been given a list of the appropriate item IDs so that they can try the application.

In the example, the application is finding all kinds of casual shirts in stock. Currently there are two kinds of casual shirt in stock, red ones and blue ones. You can also see that 63 red shirts are in stock and 20 blue ones.

Listing Items in Duke's Choice Web Application

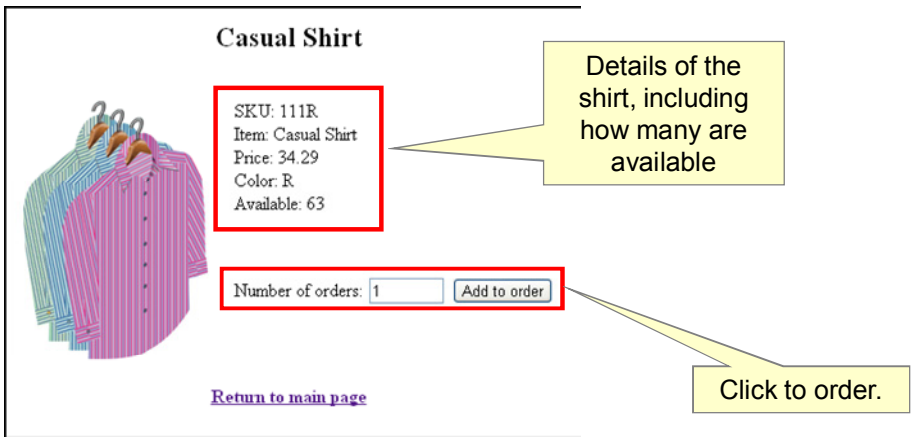


Here's the other possible application for Duke's Choice—this time a simple web application. In this case, DukesChoice.jar is copied to the application server, where it can be accessed by the UI components of the application, in this case by JSP (Java Server Pages) files.

The screenshot shows the main search page that allows customers to search for a particular item. They can pick an item from a drop-down list and all of the varieties of the item will then be listed. In the example in the slide, the list shows the same information as the command-line application: two colors of shirt and the available quantity of each.

The web application also allows a customer to click the SKU number of a particular item and, by doing so, navigate to a page that shows further details about that item.

Listing Items in Duke's Choice Web Application



This screen shows the details of the item the customer clicked. On this page, customers can add a number of shirts to their order.

The two applications shown (the command-line one and this web application) use classes very similar to the Shirt class you were introduced to right at the beginning of the course. Even though the user interface of the command-line version is very different than the web version, the item classes (Shirt, Trousers, Socks, Tent, and Fuel) are not in any way involved in the presentation of the data, so it is possible to modify any of these classes or add further classes without having to change the user interface.

Topics

- Packages
- JARs and deployment
- Two-tier and three-tier architecture
- Duke's Choice application
- Application modifications and enhancements

Enhancing the Application

- Well-designed Java software minimizes time required for:
  - Maintenance
  - Enhancements
  - Upgrades
- For Duke's Choice, it should be easy to:
  - Add new items to sell (business logic)
  - Develop new clients (presentation)
    - Take the application to a smartphone for example
  - Change the storage system (data)

ORACLE

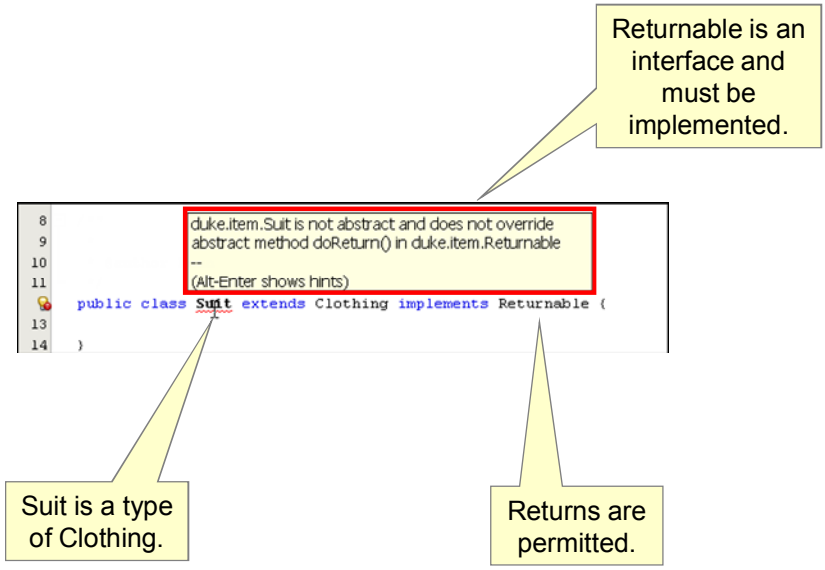
In the next sequence of slides, you will see what would be involved in adding another item class to represent a dress suit.

Adding a New Item for Sale

It is possible to add a new item for sale by:

- Extending the Clothing or Camping class or even creating a new category (for example, Books)
- Adding any new unique features for the item
- Adding some of the new items to the data store

Adding a New Item for Sale



ORACLE

NetBeans is helpful when extending abstract classes and implementing interfaces because it will give you hints about what you need to do. In the example in the slide, the new class `Suit` extends `Clothing` and implements `Returnable`. NetBeans flags that you need to implement the methods of the interface `Returnable` (in this case, the method `doReturn()`).

Implement Returnable

```
public class Suit extends Clothing implements Returnable {
    public String doReturn() {
        // In the current implementation Returnable provides
        // a marker that the item can be returned and also returns
        // a String with conditions for returning the item
        return "Suit returns must be within 3 days";
    }
}
```


The code shows a simple example of implementing the method `doReturn()` of the interface `Returnable`.

Implement Constructor

```
public class Suit extends Clothing implements Returnable {

  ...< code omitted > ...

  // Types are D = Double-breasted, S = Single-breasted, U=Unset
  private char suitType = 'U'; //

  // Constructor
  public Suit(int itemID, String description, char colorCode,
              double price,  char type, int quantityInStock) {
    super( itemID,  description,  colorCode, price, quantityInStock);
    setSuitType(type);
    setSku(getSku() + type); // To create a unique SKU
  }
}
```


This code shows the implementation of the constructor for the Suit type, given that:

- Suit has an extra attribute, suitType, that is not in the superclass Clothing
- This extra attribute is combined with the SKU (generate in the `Clothing` superclass) to create a unique SKU (Stock Keeping Unit) for this item

The Suit Class: Overriding `getDisplay()`

```
public String getDisplay(String separator) {

String displayString = "SKU: " + getSku() + separator +
"Item: " + getDescription() + separator +
"Color: " + getColorCode() + separator +
"Type: " + getSuitType() + separator +
"Price: " + getPrice() + separator +

"Available: " + getQuantityInStock();
return displayString;
}
```


The Clothing class has a `getDisplay(String separator)` method where a separator can be specified so that the attributes of the item can be written on one line and separated by a separator character, or written line by line using a newline as the separator character.

The code in the slide shows `getDisplay(String separator)` being overridden to include the suit type in the display.

```
C:\>java -jar "C:\work\Java_fundamentals\DukesChoice\dist\DukesChoice.jar" find
410
-------------------------------------------------------------------------------
SKU: 410BD | Item: Suit | Color: B | Type: D | Price: 999.99 | Available: 21
-------------------------------------------------------------------------------
SKU: 410BS | Item: Suit | Color: B | Type: S | Price: 789.99 | Available: 15
-------------------------------------------------------------------------------
SKU: 410gD | Item: Suit | Color: G | Type: D | Price: 999.99 | Available: 21
-------------------------------------------------------------------------------
SKU: 410WS | Item: Suit | Color: W | Type: S | Price: 789.99 | Available: 15
-------------------------------------------------------------------------------
```

Implement Getters and Setters

```
public class Suit extends Clothing implements Returnable {

 ...< code omitted > ...

 public char getSuitType() {
   return suitType;
 }

 public void setSuitType(char suitType) {
   if (suitType!='D' && suitType!='B') {
      throw new IllegalArgumentException("The suit type must be"
                                  + " either D = Double-breasted "
                                  + "or S = Single-breasted");       }
        this.suitType = suitType;
   }
}
```


The code shows the implementation of the getter and setter methods for the suit type. If 'D' or 'B' is not passed into the constructor, the method throws an IllegalArgumentException. Note that IllegalArgumentException is an unchecked Exception so it does not need to be thrown from this method or checked in the calling method.

Assuming it is not caught in the current implementation of the application, if an invalid argument is passed in to the method, the Duke's Choice testers will see the following:

```
C:\>java -jar
"C:\work\Java_fundamentals\DukesChoice\dist\DukesChoice.jar"
find 410
Exception in thread "main" java.lang.IllegalArgumentException:
The suit type must be either D = Double-breasted or S = Single-breasted
        at duke.item.Suit.setSuitType(Suit.java:43)
        at duke.item.Suit.<init>(Suit.java:20)
        at duke.init.DukesDB.setupDb(DukesDB.java:52)
        at duke.init.DukesDB.<init>(DukesDB.java:84)
        at duke.init.DBtest.main(DBtest.java:29)
```

Updating the Applications with the Suit Class

For the command-line application:

- Create a new DukesChoice.jar file.
- (Optional) Copy it to a new location on the file system or to another machine.

For the web application:

- Create a new DukesChoice.jar file.
- Copy it to the directory used by the application server for library files.

Note that the jar file is exactly the same in either case.

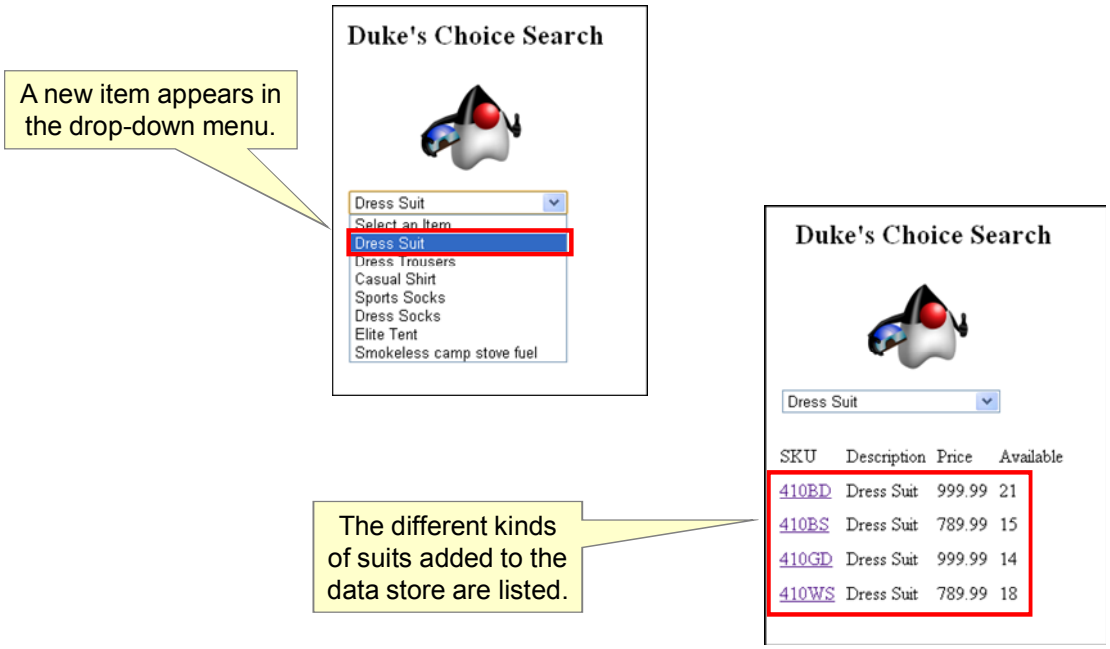
Testing the Suit Class: Command Line

```
C:\>java -jar
    "C:\work\Java_fundamentals\DukesChoice\dist\DukesChoice.jar"
    find 410
-----------------------------------------------------------------
SKU: 410BD | Item: Suit | Price: 999.99 | Color: B | Available: 21
-----------------------------------------------------------------
SKU: 410BS | Item: Suit | Price: 789.99 | Color: B | Available: 15
-----------------------------------------------------------------
SKU: 410gD | Item: Suit | Price: 999.99 | Color: G | Available: 14
-----------------------------------------------------------------
SKU: 410WS | Item: Suit | Price: 789.99 | Color: W | Available: 18
-----------------------------------------------------------------
```

ORACLE

Now the testers at Duke's Choice can search for suits in stock. However, the display doesn't let us know if the suit is single-breasted or double-breasted.

Testing the Suit Class: Web Application



ORACLE

After restarting the web application, there will be an extra item in the drop-down menu for Dress Suit, and the various kinds of Dress Suits that have been added to the data store will be listed with their SKUs.

Adding the Suit Class to the Web Application

When you click one of the Dress Suits listed, the details are displayed. Notice that since the getDisplay method was overridden, the kind of suit (S for single-breasted) is displayed. No modifications were made to the web application.

Summary

In this lesson, you should have learned how to:

- Deploy a simple application as a JAR file
- Describe the parts of a Java application including the user interface and the back end
- Describe how classes can be extended to implement new capabilities in the application

No Practice for This Lesson

This lesson has no practices.

Course Summary

In this course, you should have learned how to:

- Define the term *object* and its relationship to a class
- Demonstrate Java programming syntax
- Write a simple Java program that compiles and runs successfully
- Declare and initialize variables
- List several primitive data types
- Instantiate an object and effectively use object reference variables
- Use operators, loops, and decision constructs
- Declare and instantiate arrays and ArrayLists and be able to iterate through them

ORACLE

Java Language Quick Reference

1. **Declare a class**
```
public class Shirt{ ←class declaration
 }
```
2. **Declare a field/variable**
```
public char colorCode; ←field variable
int counter; ←local variable
```
3. **Declare and initialize a primitive variable**
```
public double price = 0.0; ←field variable
int hour = 12; ←local variable
```
4. **Declare and instantiate an object reference**
```
public ArrayList names = new ArrayList();
```
5. **Invoke a method**
```
displayInformation(); ←method with no arguments or return value
setColorCode('R'); ←method with one argument and no return value
int level = getLevel(); ←method with no arguments but returning a value
```
6. **Declare a method**
```
public void displayInformation(){…} ←method : no args, returns void
public String getName() {…} ←method: no args, returns String
public void setName(String name){…} ←method: String arg, returns void
```
7. **If/else block**
```
If (name1.equals(name2)) {
   System.out.println();
}
else {
   System.out.println("Different name.");
}
```
8. **Switch construct** ←Syntax
```
switch (variable) {
 case literal_value:
     <code_block>
     [break;]
 case another_literal_value:
     <code_block>
     [break;]
 [default:]
     <code_block>
}
```

9. Structure of a Class

```
package myClasses; ←package statement
import java.util.ArrayList; ←import statement

public class NamesList{  ←class declaration
       public ArrayList names = new ArrayList(); ←field

       public void setList(){ ←method
             // code_block;
       } ←end of method
} ←end of class
```

10. While construct ←syntax

```
while (boolean_expression) {
       // do this while expression remains true
       // code_block;
 } // end of while block
```

11. Do/while construct ←syntax

```
do {    // do the following once before evaluating expression
     // then continue to do this while expression remains true
     // code_block
}
while (boolean_expression);
```

12. For loop ←syntax

```
for (data_type init_var; boolean_expression; increment){
       // code_block;
}
```

←**example**

```
for (int i = 1; i<10; i++){
       System.out.println("Array element: " + myArray[i]);
}
```

13. Enhanced for loop ←syntax

```
for (data_type var : array_name ) {
      // code_block;
 }
```

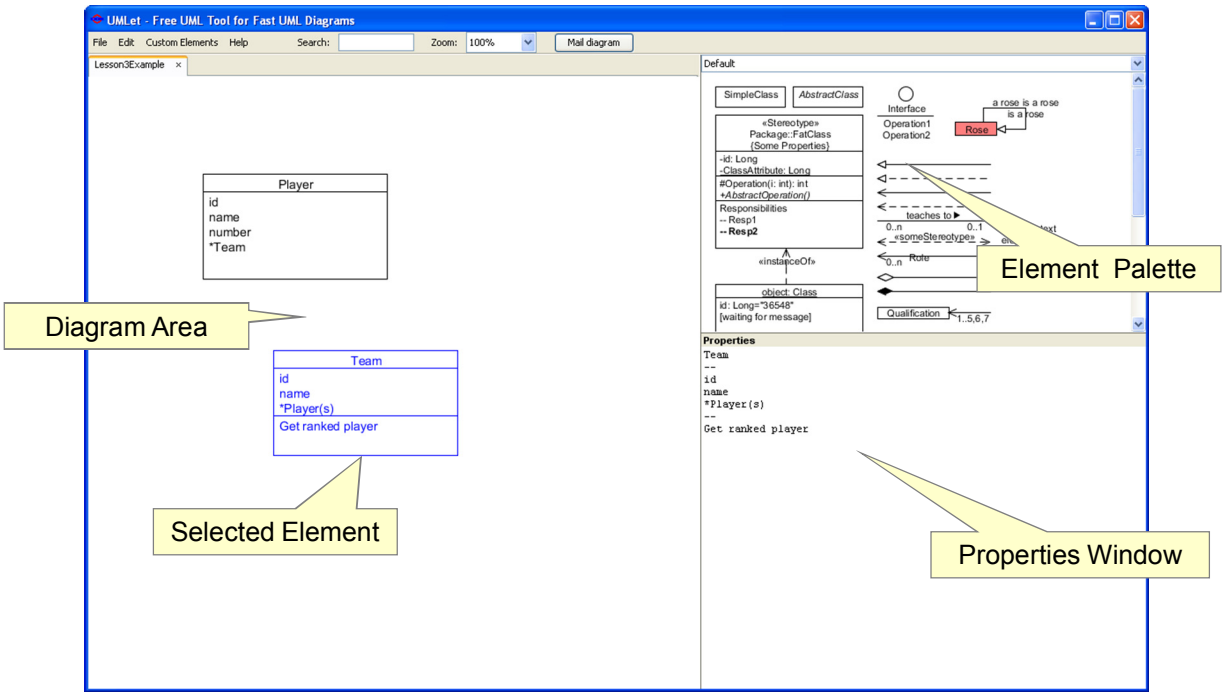
←**example**

```
for (Object obj : myList){
      System.out.println("List element: "+ obj);
}
```

B

UMLet Tips

UML Default Interface



1. **How to add elements to the diagram**
   Double-click any element in the palette; it will appear in the upper-left corner of the main diagram window.
2. **How to duplicate elements on the diagram**
   Double-click an element to duplicate it. Alternatively, you can copy and paste (or you can use their respective keyboard equivalents of Ctrl + C and Ctrl + V).
3. **How to select multiple elements**
   Press and hold Ctrl to select multiple elements.
4. **How to lasso-select multiple elements**
   Press Ctrl and click to select a rectangle containing the desired elements.
5. **How to change UML elements**
   Select an element and modify its attributes in the lower-right text panel. Each element type has a simple markup language, for example, the text “/ClassName/” causes “ClassName” to become italic. The markup languages are best explored via the sample UML elements in the palettes.

6. **How to enter comments in a UML element description**
   UMLet supports C++-style comments. Starting a line with "//" (for example, "//my comment..") enables UMLet to ignore that markup line.
7. **How to change the color of UML elements**
   Right-click an element and select its background or foreground color via the context menu.
8. Alternatively, just type the name of the color in the element description (for example, "bg=black", or "fg=red").
9. **How to create UML relations**
   Double-click a relation, and then drag its end points to the borders of UML elements; they will stick there.
10. **How to edit the relations**
    Many UML tools make it time-consuming to change the type or direction of a relation. In UMLet, simply modify the linetype, that is, by changing the line "lt=" in the element description. For example, change "lt=<." to "lt=->>" to change the direction, the arrow type, and the line's dots at the same time.
11. **How to label relations**
    Edit the name of a relation in the relation's description.
    Role names can be specified using "r1=" or "r2=".
    For multiplicities, use "m1=" or "m2=".
    Qualifiers are done with "q1=" or "q2=".
12. **How to create sequence diagrams**
    Change the current palette to "Sequence - all in one". Add the sequence diagram element to the diagram by double-clicking.
    This element's markup language is slightly more complex. The main idea is that each lane has a name and an ID (defined by the string "_name~ID_"). The IDs can then be used to define messages between lanes (for example, "id1->id3").
13. **How to create activity diagrams**
    Change the current palette to "Activity - all in one". Add the activity diagram element to the diagram by double-clicking.
    Here, TABs in the element description are used to define the activity forks.

UML Basics

The Unified Modeling Language (UML) is a graphical language for modeling software systems. The UML is not:

- A programming language: It is a set of diagrams that can be used to specify, construct, visualize, and document software designs. Software engineers use UML diagrams to construct and explain their software designs just as building architects use blueprints to construct and explain their building designs. UML has diagrams to assist in every part of application development, from requirements gathering through design, coding, testing, and deployment.
- A process for analysis and design: Its diagrams must be used with a process.

The UML was developed in the early 1990s by three leaders in the object-modeling world: Grady Booch, James Rumbaugh, and Ivar Jacobson. Their goal was to unify the major methods that they had previously developed to create a new standard for software modeling. UML is now the most commonly used modeling language. The UML specification is currently maintained by the Object Management Group (OMG), and is available on the OMG web site at http://www.omg.org/uml/.

General Elements

In general, UML diagrams represent:

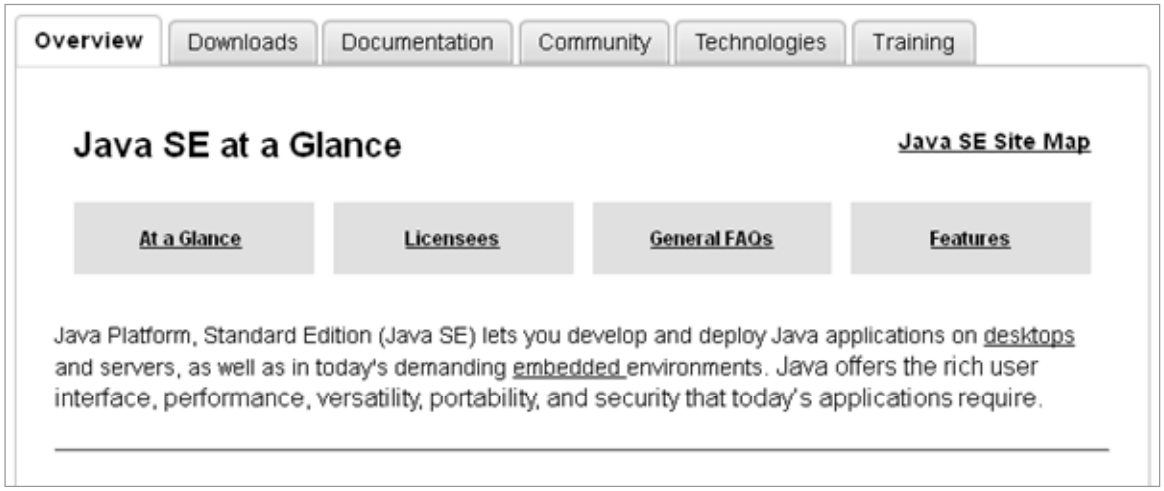
- Concepts, which are depicted as symbols (also called nodes)
- Relationships among those concepts, which are depicted as paths (also called links) that connect the symbols

These nodes and links are specialized for each particular diagram. For example, in Class diagrams, the nodes represent object classes and the links represent associations between classes and generalization (inheritance) relationships.

C

Resources

Java on Oracle Technology Network (OTN)



ORACLE

You can find many resources on the Java SE 7 pages of OTN, including:

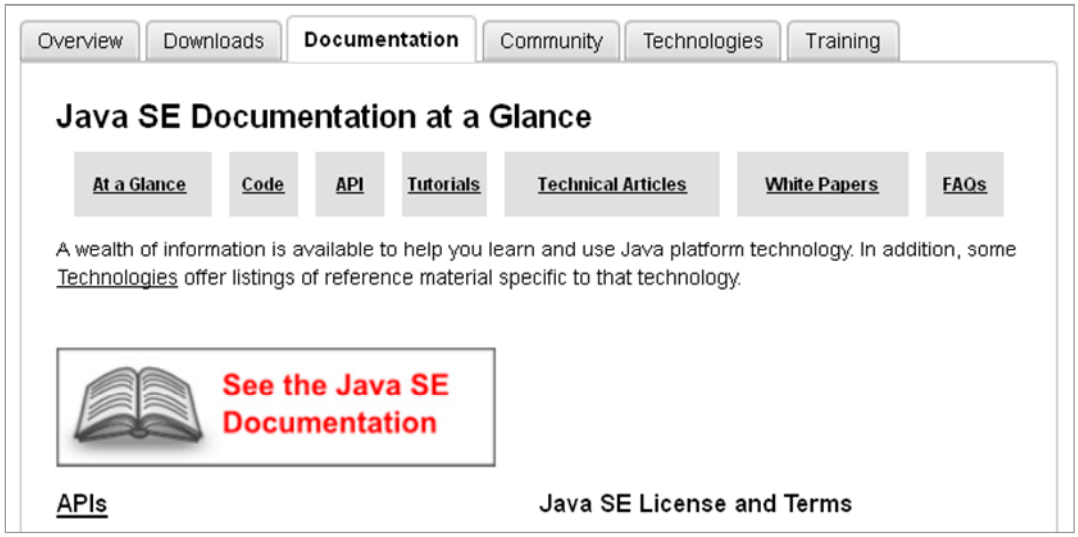
- Downloads
- Documentation
- Java Community
- Technologies
- Training

Java SE Downloads

ORACLE

The Downloads link provides the latest and previous releases for Java SE (runtime and JDK), JavaFX, JavaEE, and NetBeans.

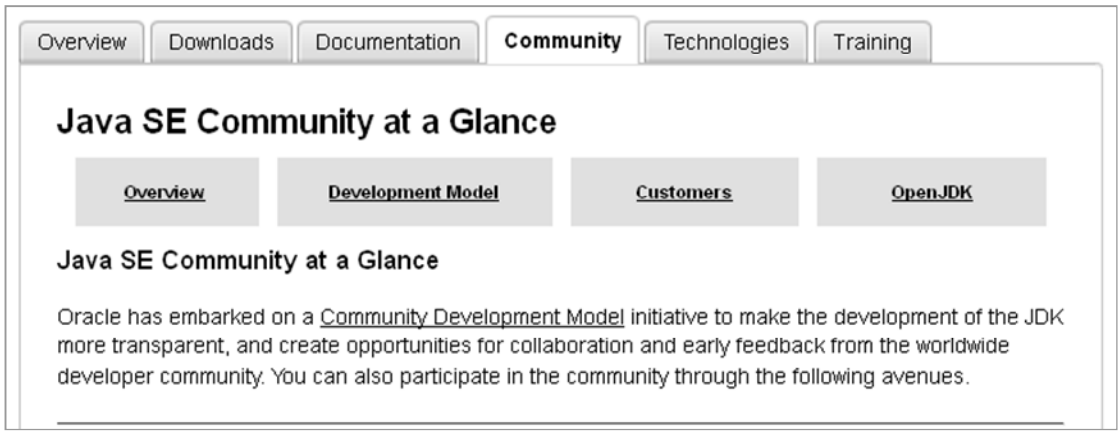
Java Documentation



You can find many resources on the Documentation page, including:

- Code
- API
- Tutorials
- Technical Articles
- White papers
- FAQs

Java Community



What is the Java Community? We frequently hear about the Java Community, as well as a variety of acronyms related to Java that you may not be familiar with, such as JUGs, JCP EC, and OpenJDK.

At a very high level, the Java Community is the term used to refer to the many individuals and organizations that develop, innovate, and use Java technology. This community includes developers as individuals, open source projects, and so on.

The Java Community page includes links to:

- **Forums**: The Java technology discussion forums are interactive message boards for sharing ideas and insights on Java technologies and programming techniques.
- **User groups**: Members of the Java User Groups meet regularly to exchange technical ideas and information.
- **Java Developer Newsletter**: The Java Developer Newsletter is a free, monthly online communication that includes news, technical articles, and events.
- **Blogs** such as the following:
  - The Java Source: Insider news from the Java team at Oracle
  - Java Oracle Blogs: Oracle blog entries tagged "java," providing wide-ranging views and information about Java technology
- **Java Developer events**
- …and more

Java Community: Expansive Reach

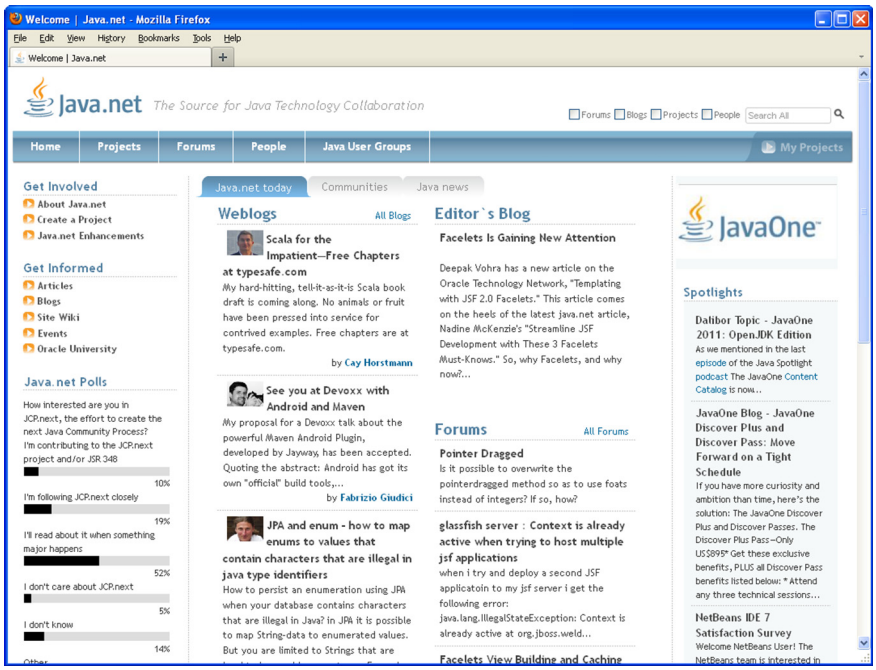
ORACLE

Forums: The Java technology discussion forums are interactive message boards for JUGs. A Java User Group (JUG) is a group of people who share a common interest in Java technology and meet on a regular basis to share technical ideas and information. The actual structure of a JUG can vary greatly—from a small number of friends and coworkers meeting informally in the evening to a large group of companies based in the same geographic area. Regardless of the size and focus of a particular JUG, the sense of community spirit remains the same.

OpenJDK (also known as **Open Java Development Kit):** A free and open source implementation of the Java programming language. In addition to Oracle, other contributors such as RedHat, IBM, and Apple all contribute to OpenJDK.

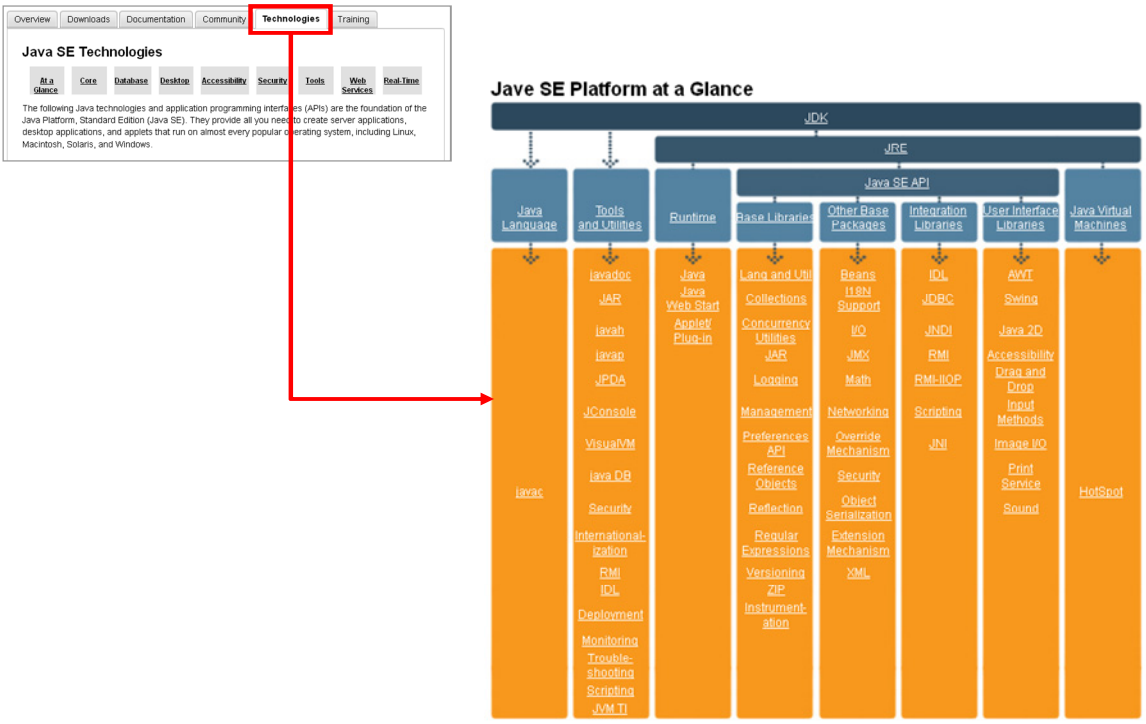
JCP: JCP stands for Java Community Process, a formalized process that allows interested parties to get involved in the definition of future versions and features of the Java platform. The JCP Executive Committee (EC) is the group of members guiding the evolution of Java technology. The EC represents both major stakeholders and a representative cross-section of the Java Community.

Java Community: Java.net



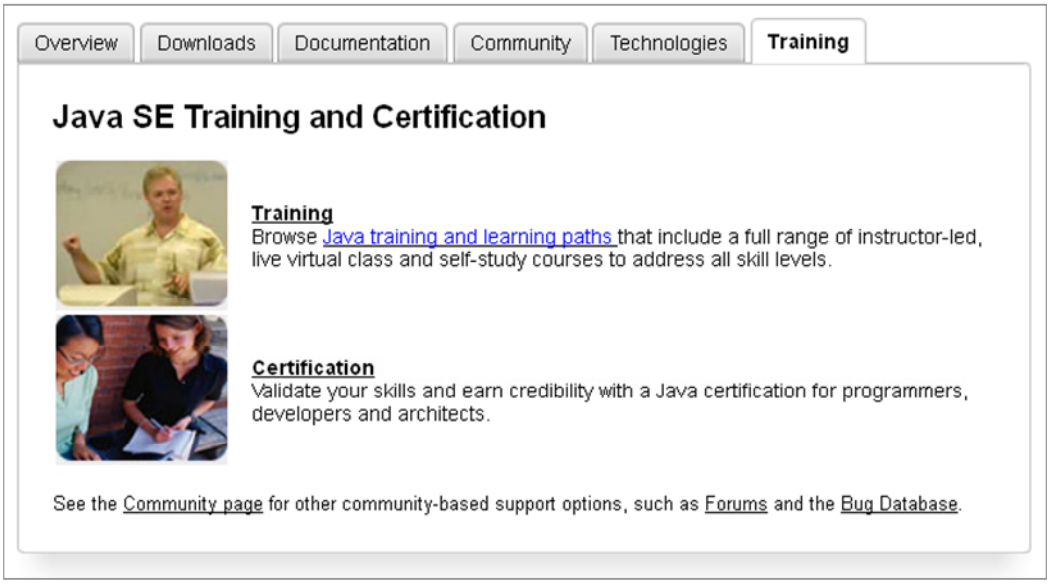
Java.net is a large community of Java developers and their projects. It welcomes anyone interested in Java, related JVM technologies, and education to the discussions and projects on the site. Java.net manages projects in a different way from most groups by maintaining curated communities of projects. That is, projects that use similar technologies or are similar types are grouped together in an area to make it easier to find other developers with similar interests and skills and their projects. The site offers technical articles, news on events, and blogs.

Java Technologies



The Java Technologies page includes a click map that describes all of the Java SE Platform technologies in detail.

Java Training

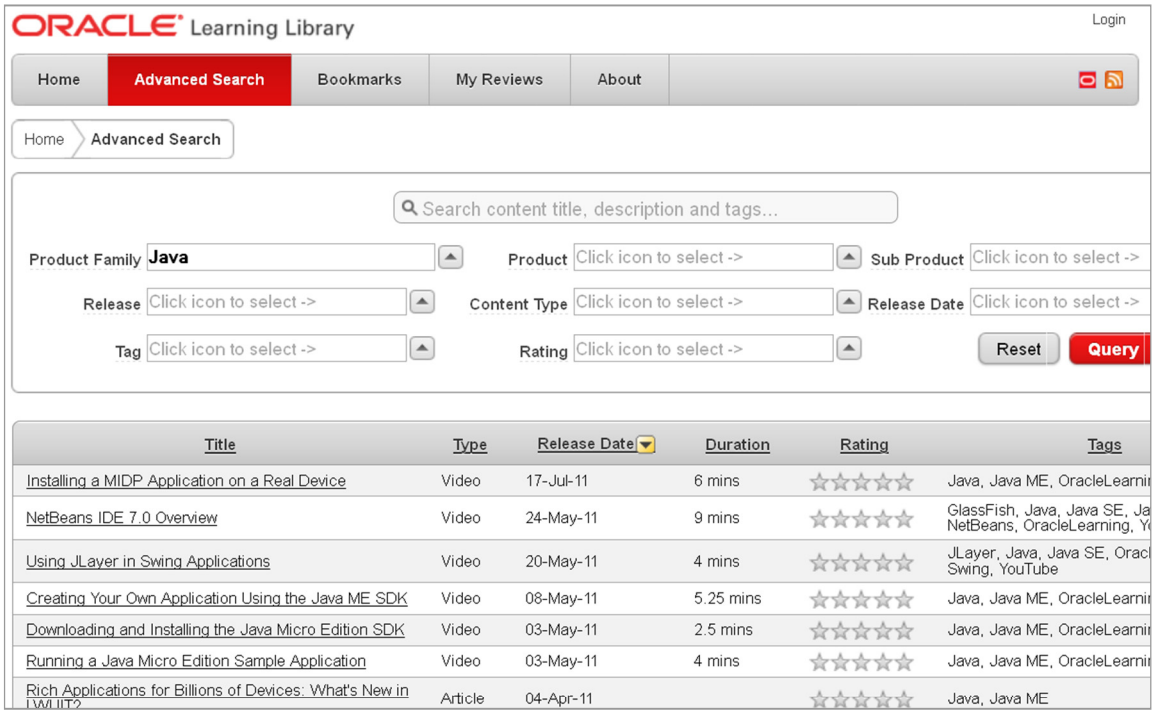


The Java SE Training and Certification page describes the available Java training as well as the Java Certification program. Oracle University offers courses that will introduce you to the Java programming language and technology so you can code smarter and develop robust programs and applications more quickly using any platform, including Oracle's application server and web infrastructure software. Validate your competency and dedication with a Java Certification—one of the most recognized credentials in the industry.

The latest Java SE training courses include:

- *Java SE 7 New Features*
- *Java Performance Tuning and Optimization*
- *Java SE 7 Fundamentals*
- *Java SE 7 Programming*

Oracle Learning Library



ORACLE

The Oracle Learning Library (OLL) features technical articles, white papers, videos, demonstrations, and Oracle by Example (OBE) tutorials on many topics, including Java. The site does require an Oracle Technology Network (OTN) login, but all content is free of charge. The OLL is available at oracle.com/oll.