Cheat Sheet

# Java For Dummies Cheat Sheet

From **Java For Dummies, 7th Edition**

By **Barry Burd**

When doing anything with Java, you need to know your Java words — those programming words, phrases, and nonsense terms that have specific meaning in the Java language, and that get it to do its thing.

## The Words in a Java Program

When you write a Java program, you can divide the program's words into several categories. This cheat sheet tells you all about those categories.

## Keywords

The Java programming language has 50 *keywords*. Each keyword has a specific meaning in the language. You can't use a keyword for anything other than its pre-assigned meaning.

The following table lists Java's keywords.

| Keyword | What It Does |
|---|---|
| abstract | Indicates that the details of a class, a method, or an interface are given elsewhere in the code. |
| assert | Tests the truth of a condition that the programmer believes is true. |
| boolean | Indicates that a value is either true or false. |
| break | Jumps out of a loop or switch. |

| | |
|---|---|
| byte | Indicates that a value is an 8-bit whole number. |
| case | Introduces one of several possible paths of execution in a switch statement. |
| catch | Introduces statements that are executed when something interrupts the flow of execution in a try clause. |
| char | Indicates that a value is a character (a single letter, digit, punctuation symbol, and so on) stored in 16 bits of memory. |
| class | Introduces a class — a blueprint for an object. |
| const | You can't use this word in a Java program. The word has no meaning but, because it's a keyword, you can't create a variable named const. |
| continue | Forces the abrupt end of the current loop iteration and begins another iteration. |
| default | Introduces a path of execution to take when no case is a match in a switch statement. |
| do | Causes the computer to repeat some statements over and over again (for instance, as long as the computer keeps getting unacceptable results). |
| double | Indicates that a value is a 64-bit number with one or more digits after the decimal point. |
| else | Introduces statements that are executed when the condition in an if statement isn't true. |
| enum | Creates a newly defined type — a group of values that a variable can have. |
| extends | Creates a subclass @@md a class that reuses functionality from a previously defined class. |
| final | Indicates that a variable's value cannot be changed, that a class's functionality cannot be extended, or that a method cannot be overridden. |
| finally | Introduces the last will and testament of the statements in a try clause. |
| float | Indicates that a value is a 32-bit number with one or more digits after the decimal point. |
| for | Gets the computer to repeat some statements over and over again (for instance, a certain number of times). |

| | |
|---|---|
| goto | You can’t use this word in a Java program. The word has no meaning. Because it’s a keyword, you can’t create a variable named goto. |
| if | Tests to see whether a condition is true. If it’s true, the computer executes certain statements; otherwise, the computer executes other statements. |
| implements | Indicates that a class provides bodies for methods whose headers are declared in an interface. |
| import | Enables the programmer to abbreviate the names of classes defined in a package. |
| instanceof | Tests to see whether a certain object comes from a certain class. |
| int | Indicates that a value is a 32-bit whole number. |
| interface | Introduces an interface. An interface is like a class but, for the most part, an interface’s methods have no bodies. |
| long | Indicates that a value is a 64-bit whole number. |
| native | Enables the programmer to use code that was written in a language other than Java. |
| new | Creates an object from an existing class. |
| package | Puts the code into a package — a collection of logically related definitions. |
| private | Indicates that a variable or method can be used only within a certain class. |
| protected | Indicates that a variable or method can be used in subclasses from another package. |
| public | Indicates that a variable, class, or method can be used by any other Java code. |
| return | Ends execution of a method and possibly returns a value to the calling code. |
| short | Indicates that a value is a 16-bit whole number. |
| static | Indicates that a variable or method belongs to a class, rather than to any object created from the class. |
| strictfp | Limits the computer’s ability to represent extra large or extra small numbers when the computer does intermediate calculations on float and double values. |

| | |
|---|---|
| super | Refers to the superclass of the code in which the word super appears. |
| switch | Tells the computer to follow one of many possible paths of execution (one of many possible cases), depending on the value of an expression. |
| synchronized | Keeps two threads from interfering with one another. |
| this | A self-reference — refers to the object in which the word this appears. |
| throw | Creates a new exception object and indicates that an exceptional situation (usually something unwanted) has occurred. |
| throws | Indicates that a method or constructor may pass the buck when an exception is thrown. |
| transient | Indicates that, if and when an object is serialized, a variable's value doesn't need to be stored. |
| try | Introduces statements that are watched (during runtime) for things that can go wrong. |
| void | Indicates that a method doesn't return a value. |
| volatile | Imposes strict rules on the use of a variable by more than one thread at a time. |
| while | Repeats some statements over and over again (as long as a condition is still true). |

## Literals

In addition to its keywords, three of the words you use in a Java program are called *literals*. Each literal has a specific meaning in the language. You can't use a literal for anything other than its pre-assigned meaning.

The following table lists Java's literal words.

| **Literal** | **What It Does** |
|---|---|
| false | One of the two values that a boolean expression can possibly have. |

| | |
|---|---|
| null | The “nothing” value. If you intend to have an expression refer to an object of some kind, but the expression doesn’t refer to any object, the expression’s value is ~~null~~. |
| true | One of the two values that a boolean expression can possibly have. |

The keywords and literal words are all called *reserved* words because each of these words is reserved for special use in the Java programming language.

## Restricted keywords

With the release of Java 9, the language has ten new words called *restricted keywords*. A restricted keyword has a specific meaning in the language, but only if you use that word in a specific way. For example, if you write

```
requires other.stuff;
```

you tell Java that your program won’t run unless it has access to some other code (the code contained in `other.stuff` ). But if you write

```
int requires = 10;
```

then `requires` is an ordinary `int` variable.

The following table lists Java’s restricted keywords.

| **Restricted Keyword** | **What It Does** |
|---|---|
| exports | Indicates that the code in a particular package is available for use by code in other modules. |
| module | A bunch of packages. |
| open | Indicates that all the packages in a module are, in a certain way, available for use by code in other modules. |
| opens | Gets access to all the code in another module. This access uses Java reflection (which tends to be messy). |
| provides | Indicates that a module makes a service available. |
| requires | Indicates that the program won’t run unless it has access to the some other code. |
| to | Names the code that has permission to use a particular piece of code. |

| | |
|---|---|
| transitive | When my code requires use of the ~~A~~ code, and the ~~Z~~ code requires use of my code, the word ~~transitive~~ means that ~~Z~~ code automatically requires ~~A~~ code. |
| uses | Indicates that a module uses a service. |
| with | Specifies a particular way of using a service. |

## Identifiers in the Java API

The Java API (Application Programming Interface) has thousands of identifiers. Each identifier is the name of something (a class, an object, a method, or something like that). These identifiers include System, out, println, String, toString, JFrame, File, Scanner, next, nextInt, Exception, close, ArrayList, stream, JTextField, Math, Random, MenuItem, Month, parseInt, Query, Rectangle, Color, Oval, paint, Robot, SQLData, Stack, Queue, TimeZone, URL, and so many others.

You can reuse any of these names for any purpose in your code. But if you do, you might have trouble using a name with its normal meaning from the Java API. For example, you can write

```
int System = 7;

java.lang.System.out.println(System);
```

But you can't write

```
int System = 7;




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

## Identifiers that you (the programmer) declare

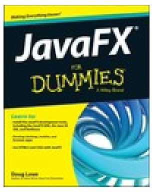
In your own Java program, you can make up names to your heart's delight. For example, in the code

```
double multiplyByTwo(double myValue) {


return myValue * 2;


}
```

the names `multiplyByTwo` and `myValue` are your very own identifiers.

When you create a new name, you can use letters, digits, underscores (_), and dollar signs ($). But don't start the name with a digit. If you try to start a name with a digit, Java replies with a "Please don't do that" message.

# JavaFX: Binding Properties



RELATED BOOK

JavaFX For Dummies

By **Doug Lowe**

JavaFX *property binding* allows you to synchronize the value of two properties so that whenever one of the properties changes, the value of the other property is updated automatically. Two types of binding are supported:

- **Unidirectional binding:** With unidirectional binding, the binding works in just one direction. For example, if you bind property A to property B, the value of property A changes when property B changes, but not the other way around.
- **Bidirectional binding:** With bidirectional binding, the two property values are synchronized so that if either property changes, the other property is automatically changed as well.

Setting up either type of binding is surprisingly easy. Every property has a `bind` and a `bindBiDirectional` method. To set up a binding, simply call this method, specifying the property you want to bind to as the argument.

Here's an example that creates a unidirectional binding on the text property of a label to the text property of a text field, so that the contents of the label always displays the contents of the text field:

```
lable1.textProperty().bind(text1.textProperty());
```

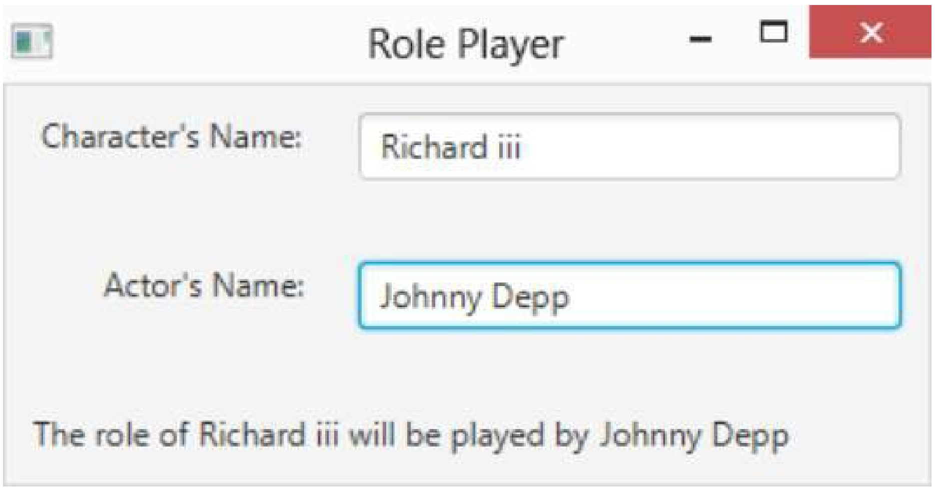
With this binding in place, the text displayed by label1 is automatically updated, character by character, when the user types data into the text field.

The following example shows how to create a bidirectional binding between two text fields, named text1 and text2:

```
text1.textProperty()
    .bindBidirectional(text2.textProperty());
```

With this binding in place, any text you type into either text field will be replicated automatically in the other.

To show how binding can be used in a complete program, this code listing shows a program with two text fields with a pair of labels bound to each. The first text field accepts the name of a character in a play, and the second text field accepts the name of an actor. The labels display the actor who will play the role, as shown in the figure.



Richard iii

Johnny Depp

The role of Richard iii will be played by Johnny Depp

```
import javafx.application.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.layout.*;
import javafx.geometry.*;
import javafx.scene.control.*;
public class RolePlayer extends Application
{
    public static void main(String[] args)
    {
        launch(args);
    }
    TextField txtCharacter;
    TextField txtActor;
    @Override public void start(Stage primaryStage)
    {
        // Create the Character label
        Label lblCharacter = new Label("Character's Name:");
        lblCharacter.setMinWidth(100);
        lblCharacter.setAlignment(Pos.BOTTOM_RIGHT);
        // Create the Character text field
        txtCharacter = new TextField();
        txtCharacter.setMinWidth(200);
        txtCharacter.setMaxWidth(200);
        txtCharacter.setPromptText("Enter the name of the character here.");
        // Create the Actor label
        Label lblActor = new Label("Actor's Name:");
        lblActor.setMinWidth(100);
        lblActor.setAlignment(Pos.BOTTOM_RIGHT);
        // Create the Actor text field
        txtActor = new TextField();
        txtActor.setMinWidth(200);
        txtActor.setMaxWidth(200);
        txtActor.setPromptText("Enter the name of the actor here.");
        // Create the Role labels
        Label lblRole1 = new Label("The role of ");
        Label lblRole2 = new Label();
        Label lblRole3 = new Label(" will be played by ");
        Label lblRole4 = new Label();
        // Create the Character pane
        HBox paneCharacter = new HBox(20, lblCharacter, txtCharacter);
        paneCharacter.setPadding(new Insets(10));
        // Create the Actor pane
        HBox paneActor = new HBox(20, lblActor, txtActor);
        paneActor.setPadding(new Insets(10));
        // Create the Role pane
        HBox paneRole = new HBox(lblRole1, lblRole2, lblRole3, lblRole4);
        paneRole.setPadding(new Insets(10));
        // Add the Character and Actor panes to a VBox
        VBox pane = new VBox(10, paneCharacter, paneActor, paneRole);
        // Create the bindings
        lblRole2.textProperty().bind(txtCharacter.textProperty());
        lblRole4.textProperty().bind(txtActor.textProperty());
        // Set the stage
        Scene scene = new Scene(pane);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Role Player");
        primaryStage.show();    }
}
```