A Guide to Using Memory Diagrams

by Mark Holliday and David Luginbuhl
March 2004

Copyright © 2004-2008
Comments, corrections, and other feedback appreciated
holliday@email.wcu.edu

This guide summarizes the memory diagram notation. A key point is that anytime execution of part of a Java statement causes a change in the state of memory, a new diagram should be drawn. Thus each statement requires at least one memory diagram and often involves several memory diagrams. For example, an assignment statement where the right hand side involves a method call requires at least three diagrams: 1) a diagram to show the method call, 2) a diagram to show the return value created by the method call, and 3) a diagram to show the assignment of the return value to the variable which is the left hand side of the assignment.

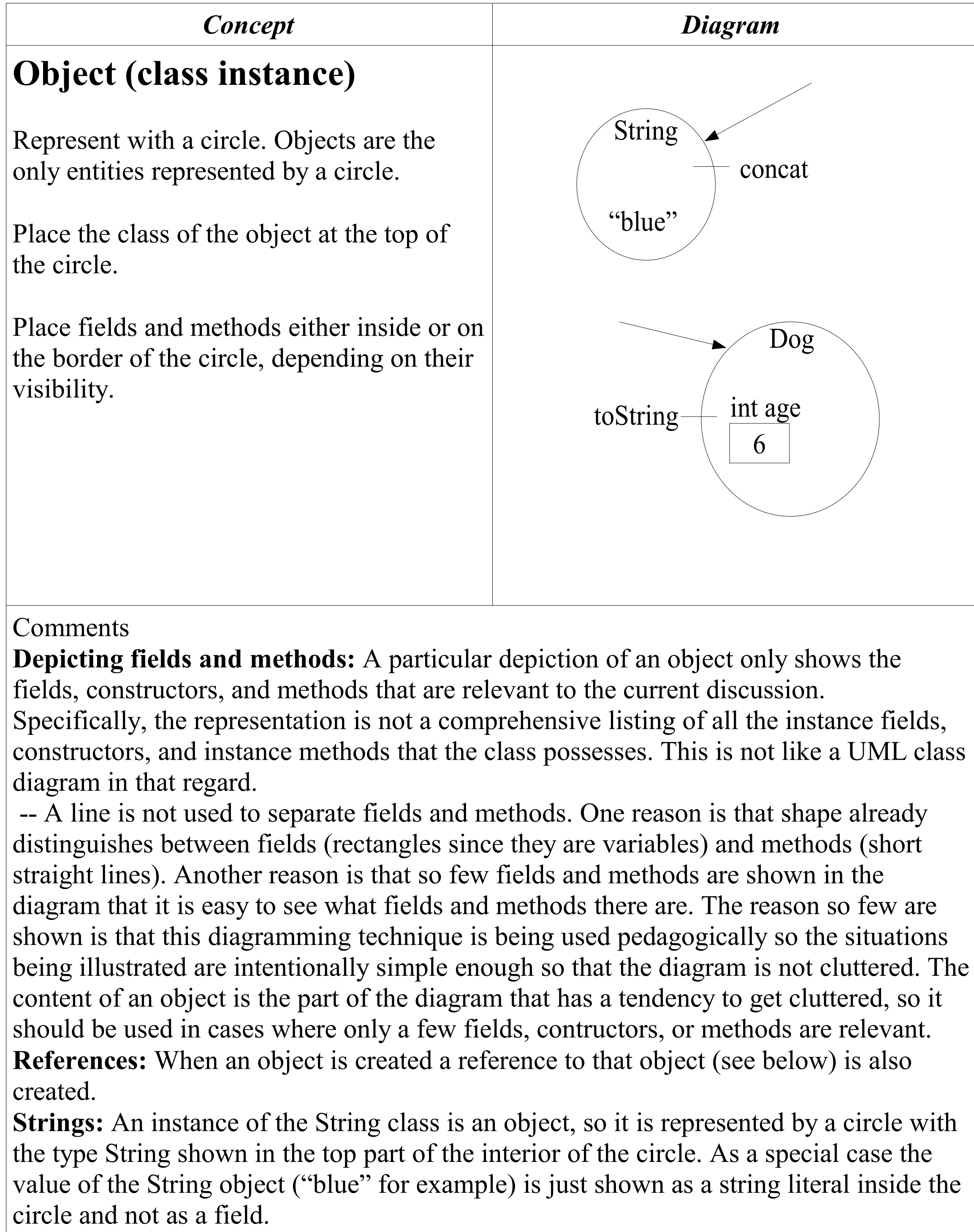
Formally, for example on a test, each diagram is drawn separately. However, in more informal settings, such as during in-class group work, instead of spending the time to redraw all the figures which is needed to create a series of separate diagrams, the presenter often shows the sequence of diagrams on top of each other. In this case, the presenter needs to indicate when the figure represents one diagram versus the next diagram.

Concept	*Diagram*
Variable (all kinds: local variables, parameters, and field (both instance fields and static fields)). Represent with a rectangle. Place value (if present) inside the rectangle. Place type and name above rectangle.	int sum Dog spot int sum 23 Dog spot double PI 3.14 Dog String name

Comments

Motivation: The rectangle represents the memory location of the variable.
Uninitialized variables: A variable that has not been initialized or has been assigned the null value is shown as empty to convey that there is nothing in the variable's memory location, although technically, primitive variables are initialized to some value by default in some cases.
Primitive variables: A primitive type variable has the actual primitive type value inside the rectangle since that is what is in the memory location.
Reference variables: A reference type variable holds the reference inside the rectangle since the reference (that is, the memory address of the start of the object being referenced) is what is inside the variable's memory location.
--A reference type variable that holds a reference must have the reference arrow pointing to the object (not to the variable) since the reference is used to get to the object (by dereferencing) and is not used to get to the variable. References are how you get to objects.
--A reference type variable that holds a reference must have the reference arrow start on the inside of the variable's rectangle, not on the border. This is to signify to the user that a variable can only hold one reference at a time. This is since the variable's memory location can only hold one memory address inside it. If the arrow just starts on the border of the rectangle, it is easy for the user to incorrectly think that several reference arrows can originate at different places on the border of a single variable. Since the

Concept	***Diagram***
arrow must start inside the rectange, when another reference arrow is placed inside the rectangle, it is clear that the reference arrow that was in the rectangle is now lost (overwritten) and must disappear. **Labeling rectangles:** The type and name of the variable are shown above the variable's rectangle with the type to the left and the name to the right. It is important that these are not inside the variable since the variable's memory location does not hold the variable's type or name. The type listed for a variable is the type used in the variable declaration. **Named constants:** If the variable is a named constant (that is, declared with the modifier final), then the border of the rectangle is made thicker. This thicker border is meant to suggest that the value of the constant cannot be changed.	

Concept	***Diagram***
Object (class instance) Represent with a circle. Objects are the only entities represented by a circle. Place the class of the object at the top of the circle. Place fields and methods either inside or on the border of the circle, depending on their visibility.	

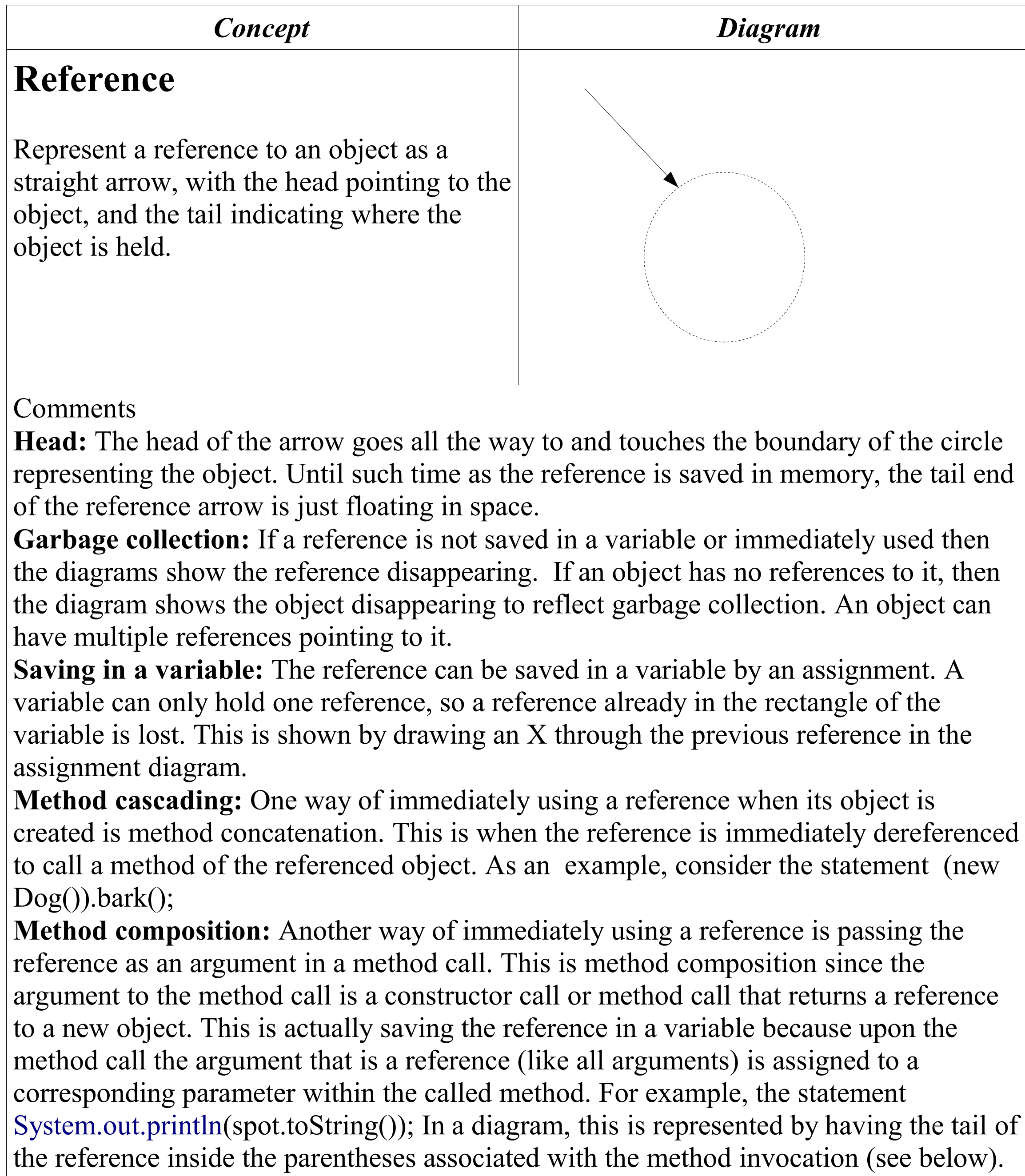
Comments

Depicting fields and methods: A particular depiction of an object only shows the fields, constructors, and methods that are relevant to the current discussion. Specifically, the representation is not a comprehensive listing of all the instance fields, constructors, and instance methods that the class possesses. This is not like a UML class diagram in that regard.

-- A line is not used to separate fields and methods. One reason is that shape already distinguishes between fields (rectangles since they are variables) and methods (short straight lines). Another reason is that so few fields and methods are shown in the diagram that it is easy to see what fields and methods there are. The reason so few are shown is that this diagramming technique is being used pedagogically so the situations being illustrated are intentionally simple enough so that the diagram is not cluttered. The content of an object is the part of the diagram that has a tendency to get cluttered, so it should be used in cases where only a few fields, contructors, or methods are relevant.

References: When an object is created a reference to that object (see below) is also created.

Strings: An instance of the String class is an object, so it is represented by a circle with the type String shown in the top part of the interior of the circle. As a special case the value of the String object ("blue" for example) is just shown as a string literal inside the circle and not as a field.

Concept	***Diagram***
Reference Represent a reference to an object as a straight arrow, with the head pointing to the object, and the tail indicating where the object is held.	

Comments

Head: The head of the arrow goes all the way to and touches the boundary of the circle representing the object. Until such time as the reference is saved in memory, the tail end of the reference arrow is just floating in space.

Garbage collection: If a reference is not saved in a variable or immediately used then the diagrams show the reference disappearing. If an object has no references to it, then the diagram shows the object disappearing to reflect garbage collection. An object can have multiple references pointing to it.

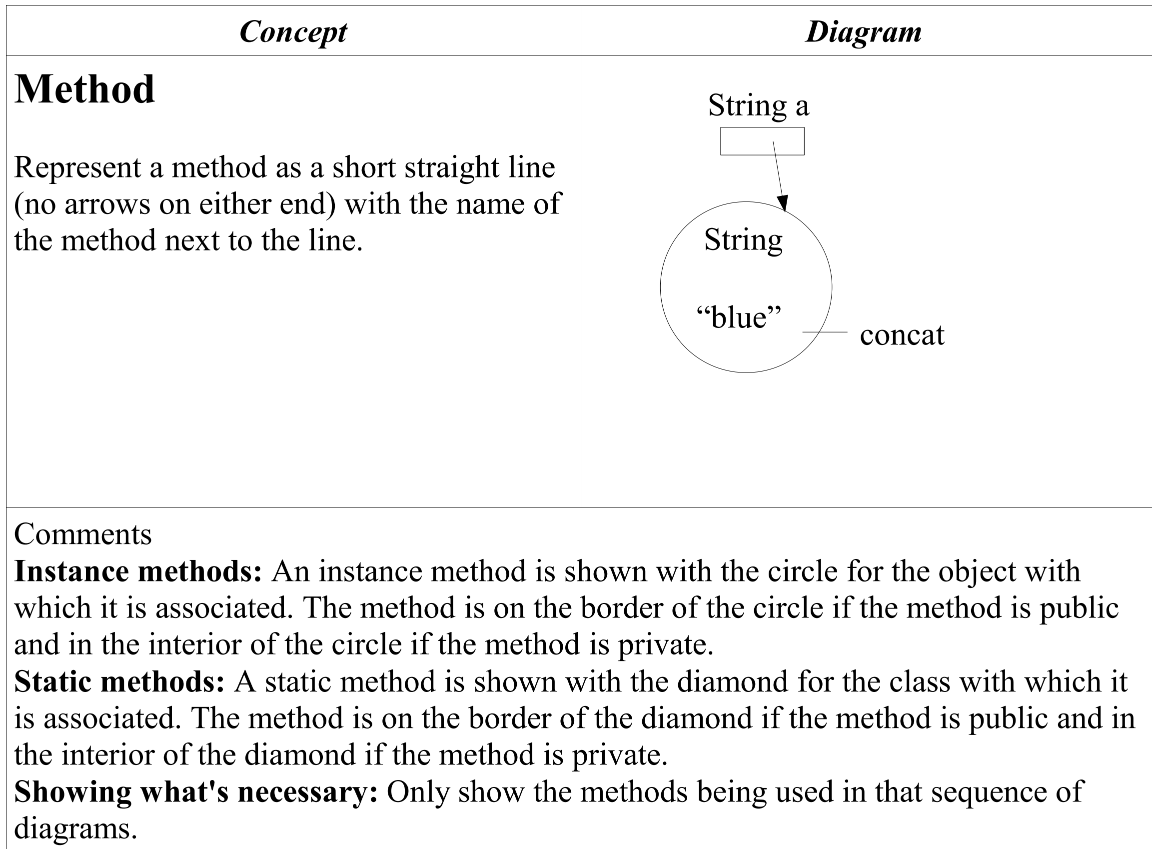
Saving in a variable: The reference can be saved in a variable by an assignment. A variable can only hold one reference, so a reference already in the rectangle of the variable is lost. This is shown by drawing an X through the previous reference in the assignment diagram.

Method cascading: One way of immediately using a reference when its object is created is method concatenation. This is when the reference is immediately dereferenced to call a method of the referenced object. As an example, consider the statement (new Dog()).bark();

Method composition: Another way of immediately using a reference is passing the reference as an argument in a method call. This is method composition since the argument to the method call is a constructor call or method call that returns a reference to a new object. This is actually saving the reference in a variable because upon the method call the argument that is a reference (like all arguments) is assigned to a corresponding parameter within the called method. For example, the statement System.out.println(spot.toString()); In a diagram, this is represented by having the tail of the reference inside the parentheses associated with the method invocation (see below).

Concept	***Diagram***
Class Represent static fields and methods by associating them with a single diamond modeling the class. Place the name of the class at the top inside of the diamond. Place fields and methods either inside or on the border of the diamond, depending on their visibility.	Math power Math.power is public System PrinntStream out System.out is public
Comments **Need for the diamond:** The diamond of a class is only shown if some static field or static method of the class is being used. **Persistence of the diamond:** If the sequence of diagrams needs the diamond for a class, then the diamond is shown in all the diagrams in the sequence since the static fields and static methods of a class exist from the start of the program execution and exist even if no instances of the class exist. For example, the diamond for the System class is needed when we are using the static public field System.out.	

Concept	***Diagram***
Visibility (public versus private) Place private fields and methods inside the circle or diamond. Place public fields and methods on the border of the circle or diamond.	Dog rover Dog toString int age 6 rover.age is private rover.toString() is public System PrinntStream out System.out is public
Comments None.	

Concept	***Diagram***
Method Represent a method as a short straight line (no arrows on either end) with the name of the method next to the line.	

Comments

Instance methods: An instance method is shown with the circle for the object with which it is associated. The method is on the border of the circle if the method is public and in the interior of the circle if the method is private.

Static methods: A static method is shown with the diamond for the class with which it is associated. The method is on the border of the diamond if the method is public and in the interior of the diamond if the method is private.

Showing what's necessary: Only show the methods being used in that sequence of diagrams.

Concept	***Diagram***
Method call Represent a method call as a wavy arrow from the reference being used to the method that is being called.	String a String b ( ) String "blue" concat String concat "red" String "bluered"

Comments

Arrow tail: The arrow originates at or near the tail of the reference that is being used to find the object which has the method that is being called. This cue is to remind the user that the reference is how the method call (that is, the wavy arrow) knows which object has the method to use (since every instance of the class in question has an instance of the method being called (assuming an instance method is being called))**.**

Arrow head: The arrow head is at the method end of the arrow.

Arguments: The wavy arrow has next to it the parentheses holding the arguments to the method call. Within the parentheses are shown the arguments. Each argument shown between the parentheses is the actual value being passed. For a primitive type argument, example values are 27, true, 2.3. For a reference type argument the passed value is a reference. The passed reference is a straight arrow that begins inside the parentheses and points to the object being referenced (the arrow head end of the arrow touches the boundary of the object being referenced).

Transience of wavy arrow: Unlike most other parts of a memory diagram, the wavy arrow of a method call is transient. It only appears in the one diagram which represents the method call. The next diagram will not have the wavy arrow since the method call (invocation) is no longer occuring.

Necessity of multiple diagrams: A method call involves two diagrams.

-- The first diagram represents the method invocation and has the wavy arrow and parentheses.

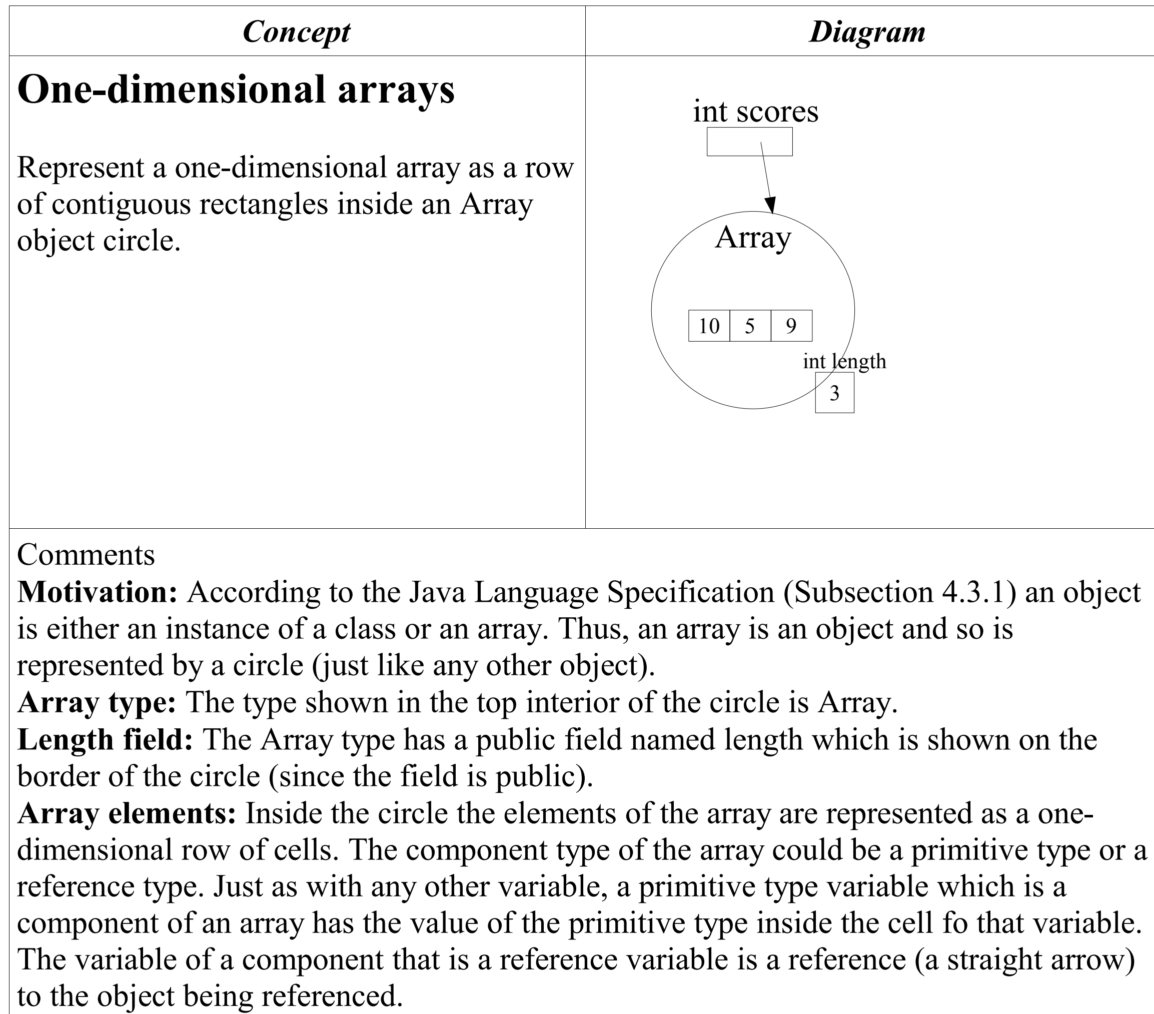
-- The second diagram represents any changes in the state of memory due to the

Concept	***Diagram***
execution of the method. --These changes may involve changes in the state of particular fields (instance or static) and may involve a return value being created and returned. A return value may be a primitive type or a reference. **--** If the return value is a primitive type, the new primitive type value is shown in the diagram.	

Concept | ***Diagram***

One-dimensional arrays

Represent a one-dimensional array as a row of contiguous rectangles inside an Array object circle.



Comments

Motivation: According to the Java Language Specification (Subsection 4.3.1) an object is either an instance of a class or an array. Thus, an array is an object and so is represented by a circle (just like any other object).
Array type: The type shown in the top interior of the circle is Array.
Length field: The Array type has a public field named length which is shown on the border of the circle (since the field is public).
Array elements: Inside the circle the elements of the array are represented as a one-dimensional row of cells. The component type of the array could be a primitive type or a reference type. Just as with any other variable, a primitive type variable which is a component of an array has the value of the primitive type inside the cell fo that variable. The variable of a component that is a reference variable is a reference (a straight arrow) to the object being referenced.

Concept	***Diagram***
Two-dimensional arrays Represent a two-dimensional array with multiple one-dimensional Array object circles, one circle to represent the first dimension, and additional circles to represent each row of the array.	int scores Array int length 2 Array int length 3 Array int length 3

Comments

Motivation: A two-dimensional array is an array of arrays. Thus, it consists of multiple objects.

First dimension: There is an object for the array for the first dimension. This is a circle just like the circle for the one-dimensional array described above. Thus, it has a public field named length.

Second dimension: The first dimension has a cell for each row of the two-dimensional array. Each of these cells (in the first dimension object) has a reference to an object that represents one row of array elements (that is, to a second dimension object).

-- This other object also is a one-dimensional array so it has a public field named length and a series of cells within representing the cells within that row.

Multi-dimensional arrays: Arrays of dimension greater than two just carry the same idea to additional levels of object. Each object is still just a one-dimensional array.

A "quick and dirty" alternative: As an approximation, when the fact that the two-dimensional array is really an array of arrays is not relevant (or has not yet been introduced), the array is shown as a single circle with a two-dimensional grid of cells within it. This diagram should only be used with caution since it is not really how memory is being used and so that fact needs to be emphasized to the students .

Concept	*Diagram*

Parameter passing

```
int sum = 3;
Dog spot = new Dog();
this.doit(sum, spot);
.
.
.
public void doit(int cnt, Dog rover)
{
.
.
.
}
```

int sum
3
Dog spot
int cnt
3
Dog rover
Dog
String name
String
“default”

Comments: Parameter passing is call-by-value in Java. Thus, an argument is an expression that is evaluated to a value that must be of the same type (of the type of a subclass) as the parameter in the called method. The parameter is a new variable local to the method that is assigned the value resulting from the expression evaluation of the argument. The name of the parameter can be the same as the name of the argument but represents a different variable (and thus a different rectangle).

To illustrate how parameter passing changes the state of memory, the memory diagrams are extended a bit in that they are integrated within the code. Thus, the diagram shows the Java code also for both the calling code and the code of the method being called. The

Concept	***Diagram***
rectangle for the variable which is the parameter is drawn next to the parameter declaration in the method header. Any variable in the calling code has its rectangle drawn next to its variable declaration in the calling code. The circles for objects are shown in a part of the diagram that can be reached by references from either the argument variable or the parameter variable.	