C6 Strings, Arrays, ArrayLists, Dates, and Lambdas

Java 8 Associate

1Z0-808

Link

https://mylearn.oracle.com/ou/exam/java-se-8-programmer-i-1z0-808/105037/110679/170387
https://docs.oracle.com/javase/specs/jls/se8/html
https://docs.oracle.com/javase/tutorial
http://www.java2s.com

https://enthuware.com
https://github.com/carlosherrerah/OCA

JUAN CARLOS HERRERA HERNANDEZ
carlos.herrera@upa.edu.mx

Contenido

>>

0. CERTIFICATION SUMMARY

CERTIFICATION OBJECTIVES

- Create and Manipulate Strings
- Manipulate Data Using the StringBuilder Class and Its Methods
- Create and Use Calendar Data
- Declare, Instantiate, Initialize, and Use a One-Dimensional Array
- Declare, Instantiate, Initialize, and Use a Multidimensional Array
- Declare and Use an ArrayList
- Use Wrapper Classes
- Use Encapsulation for Reference Variables
- Use Simple Lambda Expressions

The most important thing to remember about `String`s is that `String` objects are **immutable** «se usan una vez, permanecen en memoria para siempre», but references to `String`s are not!.

Because `StringBuilder`'s methods are not thread-safe, they tend to run faster than `StringBuffer` methods, so choose `StringBuilder` whenever threading is not an issue. Both `StringBuffer` and `StringBuilder` objects can have their value changed over and over without your having to create new objects.

Similar to `String`s, all of the **calendar classes** we studied create immutable objects. In addition, these classes use factory methods exclusively to create new objects. The keyword `new` cannot be used with these classes.

`ArrayList`s are like arrays with superpowers that allow them to grow and shrink dynamically and to make it easy for you to insert and delete elements at locations of your choosing within the list. `ArrayList`s cannot hold primitives, and that if you want to make an `ArrayList` filled with a given type of primitive values, you use **"wrapper"** classes to turn a primitive value into an object that represents that value. "autoboxing".

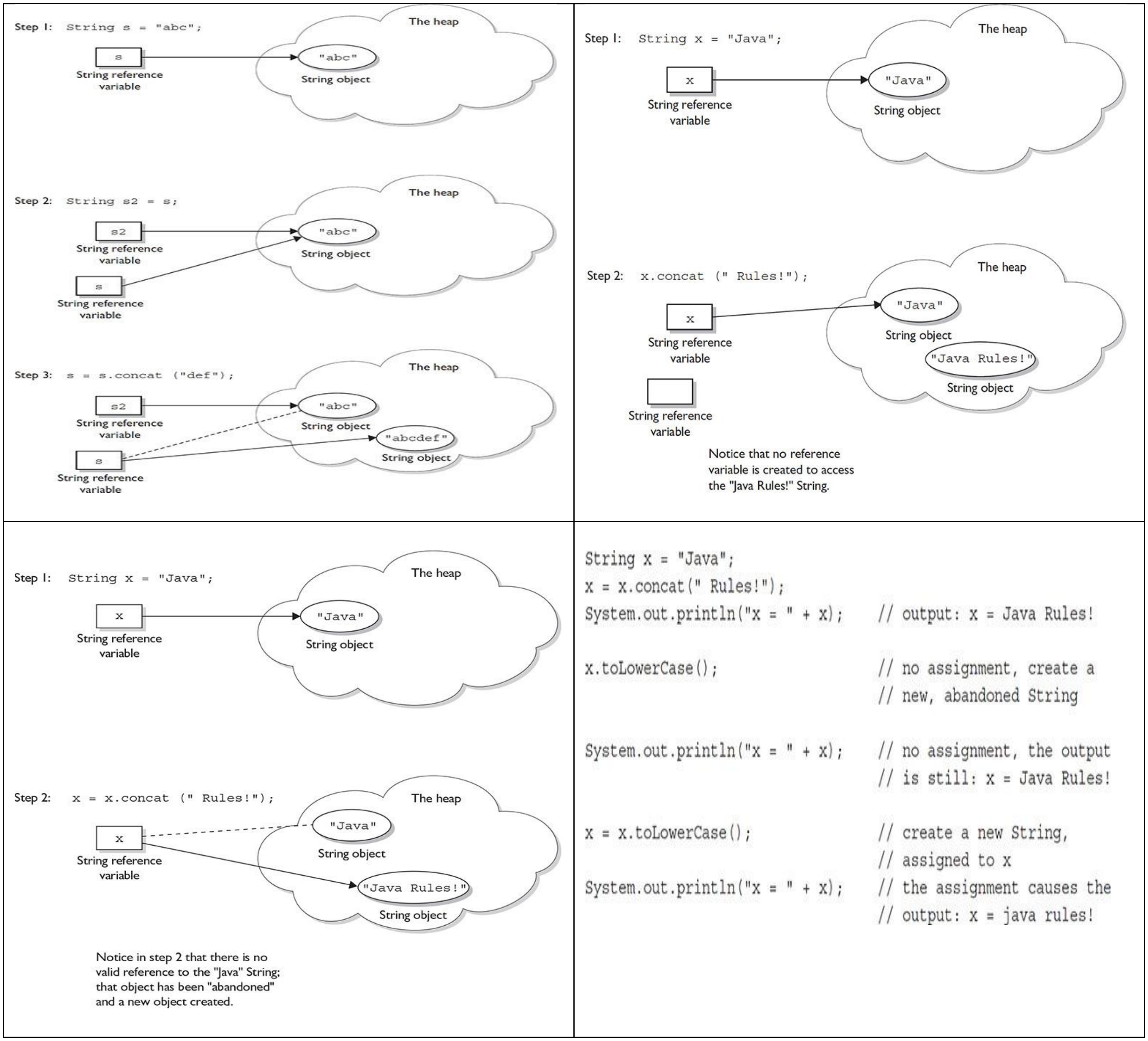
The basic idea of lambdas is that you can pass a bit of code from one method to another. The `Predicate` interface is one of many "functional interfaces" provided in the Java 8 API.

>>

1. Using String and StringBuilder (OCA Objectives 9.2 and 9.1)

- `String` objects are immutable, and `String` reference variables are not.
- If you create a new `String` without assigning it, it will be lost to your program.
- If you redirect a `String` reference to a new `String`, the old `String` can be lost.
- `String` methods use zero-based indexes, except for the second argument of `substring()`.
- The `String` class is `final`—it cannot be extended.
- When the JVM finds a `String` literal, it is added to the `String` literal pool.
- Strings have a *method* called `length()`—arrays have an *attribute* named `length`.
- `StringBuilder` objects are mutable—they can change without creating a new object.
- `StringBuilder` methods act on the invoking object, and objects can change without an explicit assignment in the statement.
- Remember that chained methods are evaluated from left to right.
- `String` methods to remember: `charAt()`, `concat()`, `equalsIgnoreCase()`, `length()`, `replace()`, `substring()`, `toLowerCase()`, `toString()`, `toUpperCase()`, and `trim()`.
- `StringBuilder` methods to remember: `append()`, `delete()`, `insert()`, `reverse()`, and `toString()`.

1.1. The String Class


Step 1: String s = "abc";
s
String reference variable
The heap
"abc"
String object
Step 2: String s2 = s;
s2
String reference variable
s
String reference variable
The heap
"abc"
String object
Step 3: s = s.concat ("def");
s2
String reference variable
s
String reference variable
The heap
"abc"
String object
"abcdef"
String object
Step 1: String x = "Java";
x
String reference variable
The heap
"Java"
String object
Step 2: x.concat (" Rules!");
x
String reference variable
String reference variable
The heap
"Java"
String object
"Java Rules!"
String object
Notice that no reference variable is created to access the "Java Rules!" String.
Step 1: String x = "Java";
x
String reference variable
The heap
"Java"
String object
Step 2: x = x.concat (" Rules!");
x
String reference variable
The heap
"Java"
String object
"Java Rules!"
String object
Notice in step 2 that there is no valid reference to the "Java" String; that object has been "abandoned" and a new object created.

```
String x = "Java";
x = x.concat(" Rules!");
System.out.println("x = " + x);   // output: x = Java Rules!

x.toLowerCase();                  // no assignment, create a
                                  // new, abandoned String

System.out.println("x = " + x);   // no assignment, the output
                                  // is still: x = Java Rules!

x = x.toLowerCase();              // create a new String,
                                  // assigned to x
System.out.println("x = " + x);   // the assignment causes the
                                  // output: x = java rules!
```

Figure 6-1 `String` objects and their reference variables

<table>
<tr>
<td>String s1 = "spring ";
String s2 = s1 + "summer ";
s1.concat("fall ");
s2.concat(s1);
s1 += "winter ";
System.out.println(s1 + " : " + s2);</td>
<td>What is the output?
how many String objects and
how many reference variables were created prior to the println statement?
two reference variables: <code>s1</code> and <code>s2</code>
A total of eight <code>String</code> objects were created</td>
</tr>
<tr>
<td colspan="2"><code>"spring "</code>, <code>"summer "</code> (lost), <code>"spring summer "</code>, <code>"fall "</code> (lost), <code>"spring fall "</code> (lost), <code>"spring summer spring "</code> (lost), <code>"winter "</code> (lost), <code>"spring winter "</code> (at this point <code>"spring "</code> is lost)</td>
</tr>
</table>

Important Methods in the String Class

The following methods are some of the more commonly used methods in the `String` class, and they are also the ones you're most likely to encounter on the exam.

charAt() Returns the character located at the specified index

concat() Appends one string to the end of another (+ also works)

equalsIgnoreCase() Determines the equality of two strings, ignoring case

length() Returns the number of characters in a string

replace() Replaces occurrences of a character with a new character

substring() Returns a part of a string

toLowerCase() Returns a string, with uppercase characters converted to lowercase

toString() Returns the value of a string

toUpperCase() Returns a string, with lowercase characters converted to uppercase

trim() Removes whitespace from both ends of a string

public char charAt(int index)

public String concat(String s)

public boolean equalsIgnoreCase(String s)

public int length()

public String replace(char old, char new)

public String substring(int begin) and

public String substring(int begin, int end)

public String toLowerCase()

public String toString()

public String toUpperCase()

public String trim()

```
    //         01234567890
    String x ="UNIVERSIDAD";
    System.out.println(x.substring(6, 10));  // no incluye el 10

    System.out.println(x.replace("SIDA", "vih"));
```

1.2. The StringBuilder Class

The `java.lang.StringBuilder` class should be used when you have to make a lot of modifications to strings of characters. **Prefer StringBuilder to StringBuffer.**

Important Methods in the StringBuilder Class

public StringBuilder append(String s)
public StringBuilder delete(int start, int end)
public StringBuilder insert(int offset, String s)
public StringBuilder reverse()
public String toString()

“chained methods.”

```
result = method1().method2().method3();
```

```
StringBuilder sb = new StringBuilder("123456789");
sb.delete(0, 3);  // sin tocar el limite superior
```

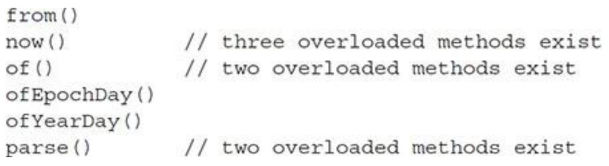
>>

2. Working with Calendar Data (OCA Objective 9.3)

- On the exam all the objects created using the calendar classes are immutable, but their reference variables are not.
- If you create a new calendar object without assigning it, it will be lost to your program.
- If you redirect a calendar reference to a new calendar object, the old calendar object can be lost.
- All of the objects created using the exam's calendar classes must be created using factory methods (e.g., `from()`, `now()`, `of()`, `parse()`); the keyword **`new`** is not allowed.
- The `until()` and `between()` methods perform complex calculations that determine the amount of time between the values of two calendar objects.
- The `DateTimeFormatter` class uses the `parse()` method to parse input Strings into valid calendar objects.
- The `DateTimeFormatter` class uses the `format()` method to format calendar objects into beautifully formed Strings.

2.1. Factory Classes

Usually, when a class has no public constructors and provides at least one `public static` method that can create new instances of the class, that class is called a *factory class,* and any method that is invoked to get a new instance of the class is called a *factory method.* If we use the `LocalDate` class as an example, we find the following `static` methods that create and return a new instance:



```
from()
now()          // three overloaded methods exist
of()           // two overloaded methods exist
ofEpochDay()
ofYearDay()
parse()        // two overloaded methods exist
```

Remember the exam's date and time classes use factory methods to create new objects.

2.2. Using and Manipulating Dates and Times

```
LocalDate ld = LocalDate.now();
Period p = Period.of(1, 2, 3);              // 1 year, 2 months, 3 days P1Y2M3D

System.out.println(ld.getDayOfWeek());             // name of the day
System.out.println(ld.getDayOfWeek().getValue());  // [1..7] Monday to Sunday
System.out.println(ld.plus(p));
```

2.3. Formatting Dates and Times

```
//DateTimeFormatter DTF = DateTimeFormatter.ofPattern("dd/MMM/yyyy");
  DateTimeFormatter DTF = DateTimeFormatter.ofPattern("E MMM dd, yyyy");
  LocalDate ld = LocalDate.now();
  System.out.println(ld.format(DTF));
```

>>

3. Using Arrays (OCA Objectives 4.1 and 4.2)

- Arrays can hold primitives or objects, but the array itself is always an object.
- When you declare an array, the brackets can be to the left or right of the name.
- It is never legal to include the size of an array in the declaration.
- You must include the size of an array when you construct it (using `new`) unless you are creating an anonymous array.
- Elements in an array of objects are not automatically created, although primitive array elements are given default values.
- You'll get a `NullPointerException` if you try to use an array element in an object array if that element does not refer to a real object.
- Arrays are indexed beginning with zero.
- An `ArrayIndexOutOfBoundsException` occurs if you use a bad index value.
- Arrays have a `length` attribute whose value is the number of array elements.
- The last index you can access is always one less than the length of the array.
- Multidimensional arrays are just arrays of arrays.
- The dimensions in a multidimensional array can have different lengths.
- An array of primitives can accept any value that can be promoted implicitly to the array's declared type—for example, a `byte` variable can go in an `int` array.
- An array of objects can hold any object that passes the IS-A (or `instanceof`) test for the declared type of the array. For example, if `Horse` extends `Animal`, then a `Horse` object can go into an `Animal` array.

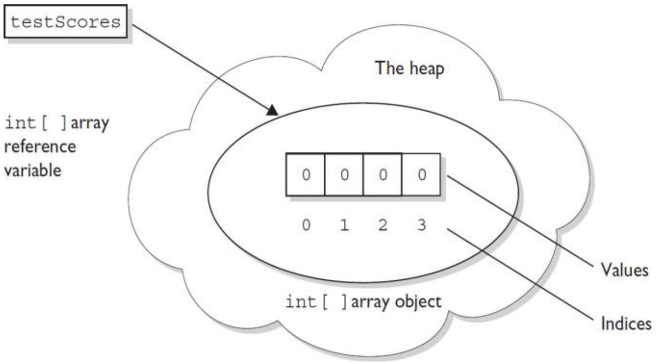
- If you assign an array to a previously declared array reference, the array you’re assigning must be the same dimension as the reference you’re assigning it to.

- You can assign an array of one type to a previously declared array reference of one of its supertypes. For example, a `Honda` array can be assigned to an array declared as type `Car` (assuming `Honda` extends `Car`).

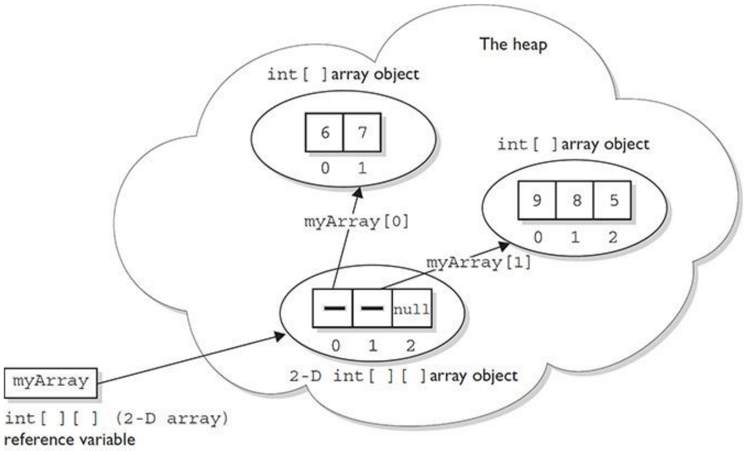
Arrays are objects in Java that store multiple variables of the same type. Arrays can hold either primitives or object references, but the array itself will always be an object on the heap, even if the array is declared to hold primitive elements.

you need to know three things:
How to make an array reference variable (declare)
How to make an array object (construct)
How to populate the array with elements (initialize)

3.1. Declaring an Array

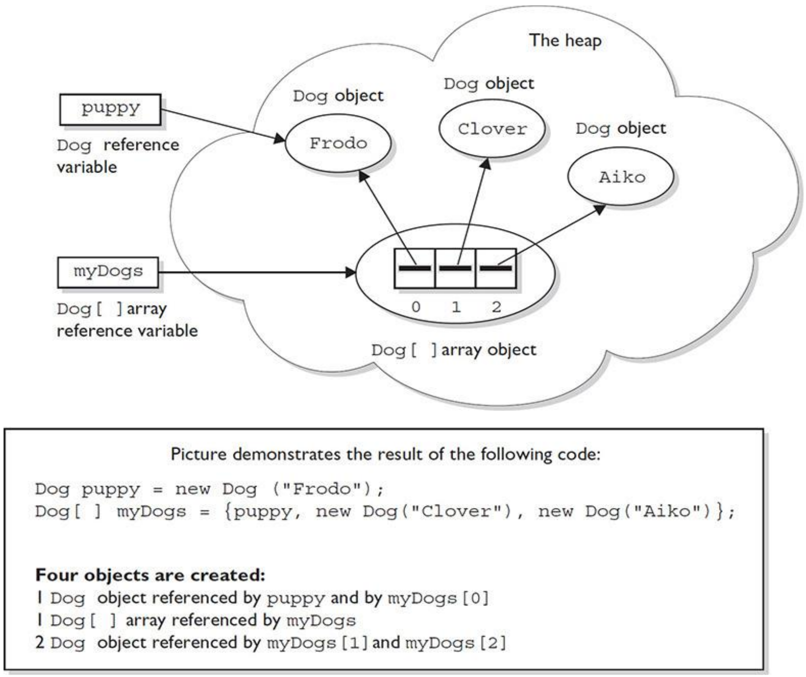


int[] array = new int[4]



Picture demonstrates the result of the following code:

```
int[ ][ ] myArray = new int[3][ ];
myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

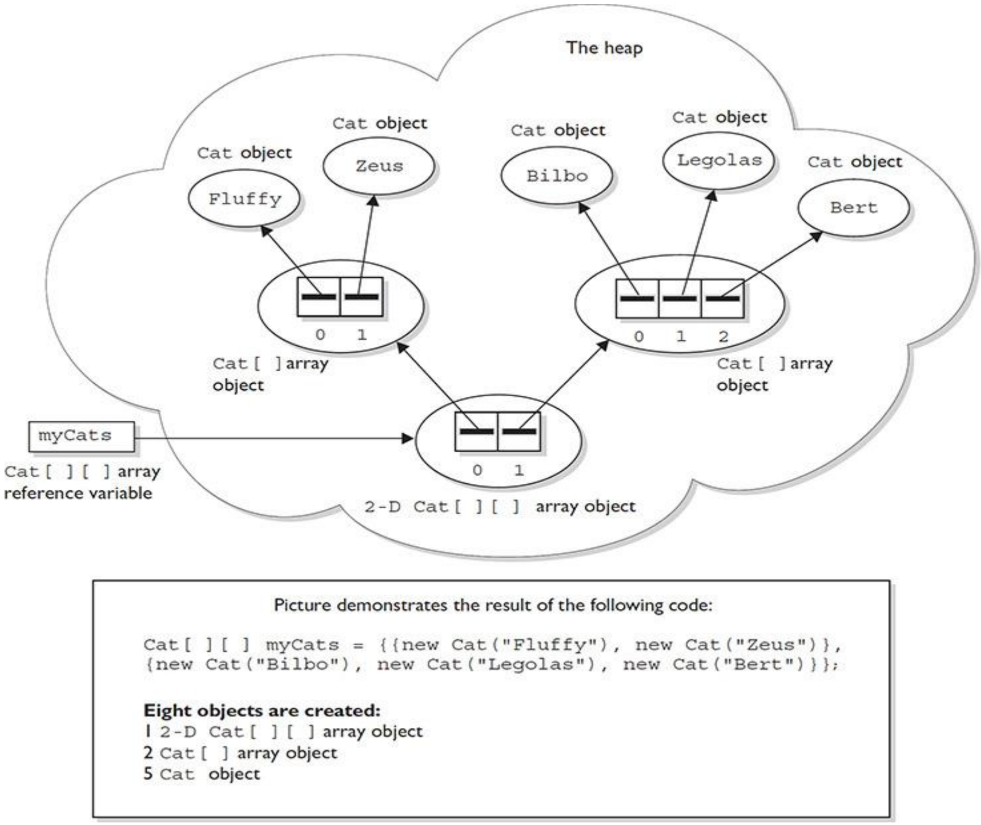


Picture demonstrates the result of the following code:

```
Dog puppy = new Dog ("Frodo");
Dog[ ] myDogs = {puppy, new Dog("Clover"), new Dog("Aiko")};
```

Four objects are created:
1 Dog object referenced by puppy and by myDogs[0]
1 Dog[ ] array referenced by myDogs
2 Dog object referenced by myDogs[1] and myDogs[2]

Figure 6-6 Declaring, constructing, and initializing an array of objects

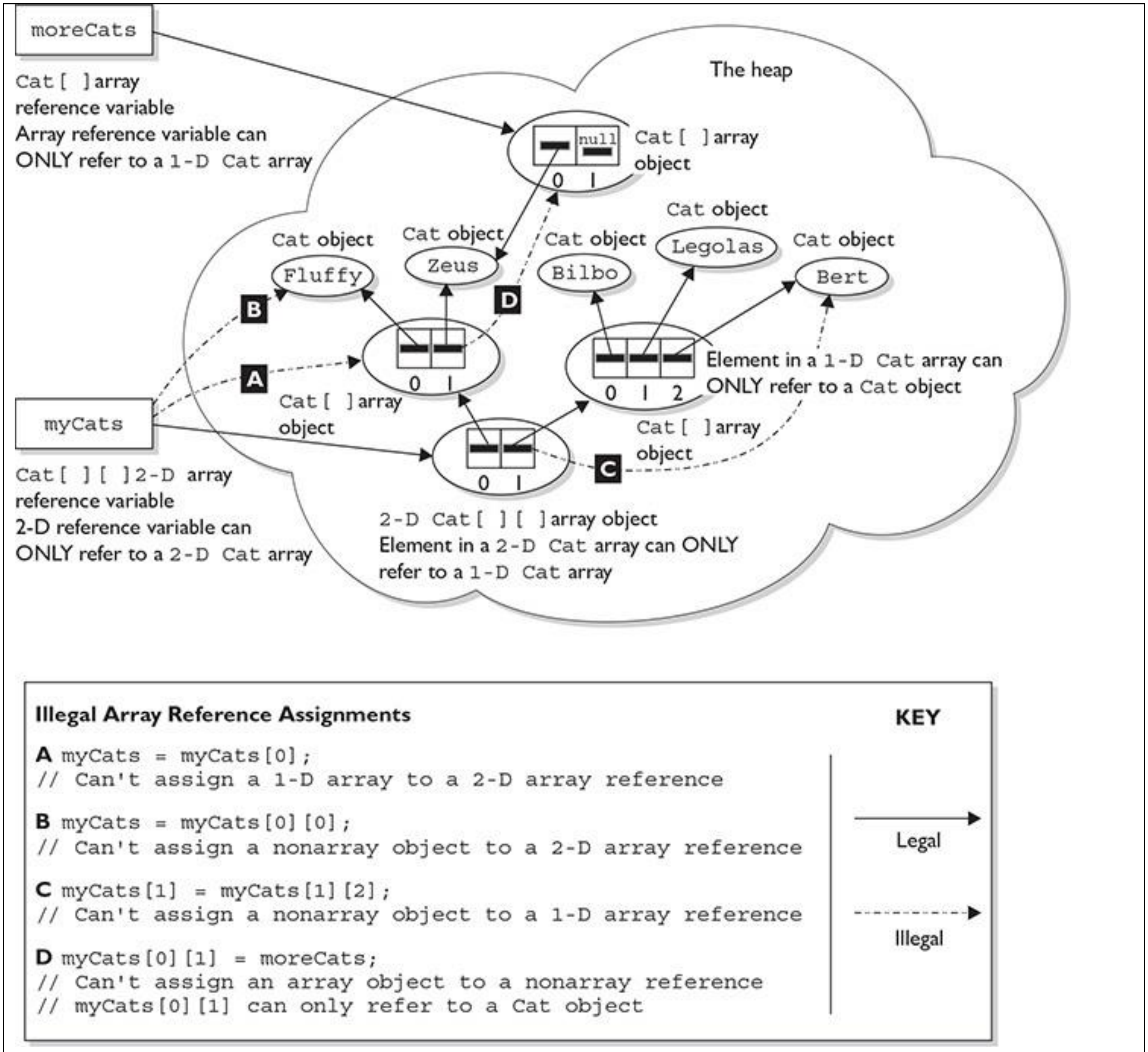


Picture demonstrates the result of the following code:

```
Cat[ ][ ] myCats = {{new Cat("Fluffy"), new Cat("Zeus")},
{new Cat("Bilbo"), new Cat("Legolas"), new Cat("Bert")}};
```

Eight objects are created:
1 2-D Cat[ ][ ] array object
2 Cat[ ] array object
5 Cat object

Figure 6-7 Declaring, constructing, and initializing a two-dimensional array



Illegal Array Reference Assignments

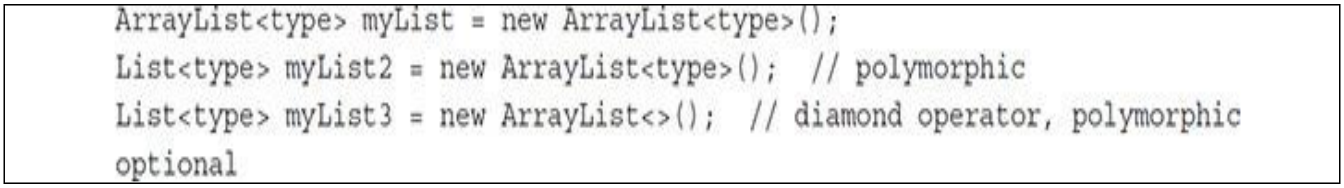
```
A myCats = myCats[0];
// Can't assign a 1-D array to a 2-D array reference

B myCats = myCats[0][0];
// Can't assign a nonarray object to a 2-D array reference

C myCats[1] = myCats[1][2];
// Can't assign a nonarray object to a 1-D array reference

D myCats[0][1] = moreCats;
// Can't assign an array object to a nonarray reference
// myCats[0][1] can only refer to a Cat object
```

Figure 6-8 Legal and illegal array assignments

>>

4. Using ArrayLists and Wrappers (OCA Objectives 9.4 and 2.5)

- `ArrayList`s allow you to **resize** your list and make insertions and deletions to your list far more easily than arrays.
- `ArrayList`s are ordered by default. When you use the `add()` method with no index argument, the new entry will be appended to the end of the `ArrayList`.
- For the OCA 8 exam, the only `ArrayList` declarations you need to know are of this form:



```
ArrayList<type> myList = new ArrayList<type>();
List<type> myList2 = new ArrayList<type>();  // polymorphic
List<type> myList3 = new ArrayList<>();  // diamond operator, polymorphic
optional
```

- `ArrayList`s can hold only objects, **not primitives**, but remember that **autoboxing** can make it look like you're adding primitives to an `ArrayList` when, in fact, you're adding a **wrapper** object version of a primitive.

- An `ArrayList`'s index starts at 0.

- `ArrayList`s can have **duplicate** entries. Note: Determining whether two objects are duplicates is trickier than it seems and doesn't come up until the OCP 8 exam.

- `ArrayList` methods to remember: `add(element)`, `add(index, element)`, `clear()`, `contains(object)`, `get(index)`, `indexOf(object)`, `remove(index)`, `remove(object)`, and `size()`.

The Java API provides an extensive range of classes that support common data structures such as `List`s, `Set`s, `Map`s, *and* `Queue`s. "The Collection API"

When to Use ArrayLists

- You need to be able to <u>increase</u> and decrease the size of your list of things.
- The order of things in your list is important and might change.

```
String[] cities = new String[3];
List<String> cities = new ArrayList<String>();
```

Important Methods in the ArrayList Class

The following methods are some of the more commonly used methods in the `ArrayList` class and also those that you're most likely to encounter on the exam:

- **add(element)** Adds this element to the **end** of the `ArrayList`
- **add(index, element)** Adds this element at the index point and shifts the remaining elements back (for example, what was at `index` is now at `index + 1`)
- **clear()** Removes all the elements from the `ArrayList`
- **boolean contains(element)** Returns whether the `element` is in the list
- **Object get(index)** Returns the `Object` located at `index`
- **int indexOf(Object)** Returns the (`int`) location of the element or `-1` if the `Object` is not found
- **remove(index)** Removes the element at that `index` and shifts later elements toward the beginning one space
- **remove(Object)** Removes the **first** occurrence of the `Object` and shifts later elements toward the beginning one space
- **int size()** Returns the number of elements in the `ArrayList`

Autoboxing with ArrayLists

In general, collections like `ArrayList` can hold objects but not primitives.
Prior to Java 5, a common use for the so-called wrapper classes (e.g., `Integer`, `Float`, `Boolean`, and so on).

wrapper objects are immutable…

All the wrapper classes except Character provide <u>two constructors</u>:
one takes a primitive of the type being constructed,
and the other takes a String representation of the type being constructed.
For example,

```
    Integer x1= new Integer(5);
    Integer x2= new Integer("5");
```

are both valid ways to construct a new Integer object (that "wraps" the value 5).

In order to **save memory**, two instances of the following wrapper objects (created through **boxing**) will always be `==` when their primitive values are the same:

- `Boolean`
- `Byte`
- `Character` from `\u0000` to `\u007f` (`7f` is `127` in decimal)
- `Short` and `Integer` from `-128` to `127`   * * * * * * * * * * * * * *

When == is used to compare a primitive to a wrapper, the wrapper will be unwrapped and the comparison will be primitive to primitive.

```
    System.out.println("------> Integer cache");
    Integer i1 = 10;
    Integer i2 = 10;
    if (i1 != i2) System.out.println("diferentes Objectos");
    if (i1.equals(i2)) System.out.println("mismo valor");
    Integer i3 = 1000;
    Integer i4 = 1000;
    if (i3 == i4) System.out.println("mismo objeto");
    if (i3.equals(i4)) System.out.println("mismo valor");
    System.out.println("<------");
```

4.1. The Java 7 "Diamond" Syntax

La notación de diamante simplifica el código, reduce la redundancia y mejora la legibilidad y el mantenimiento.

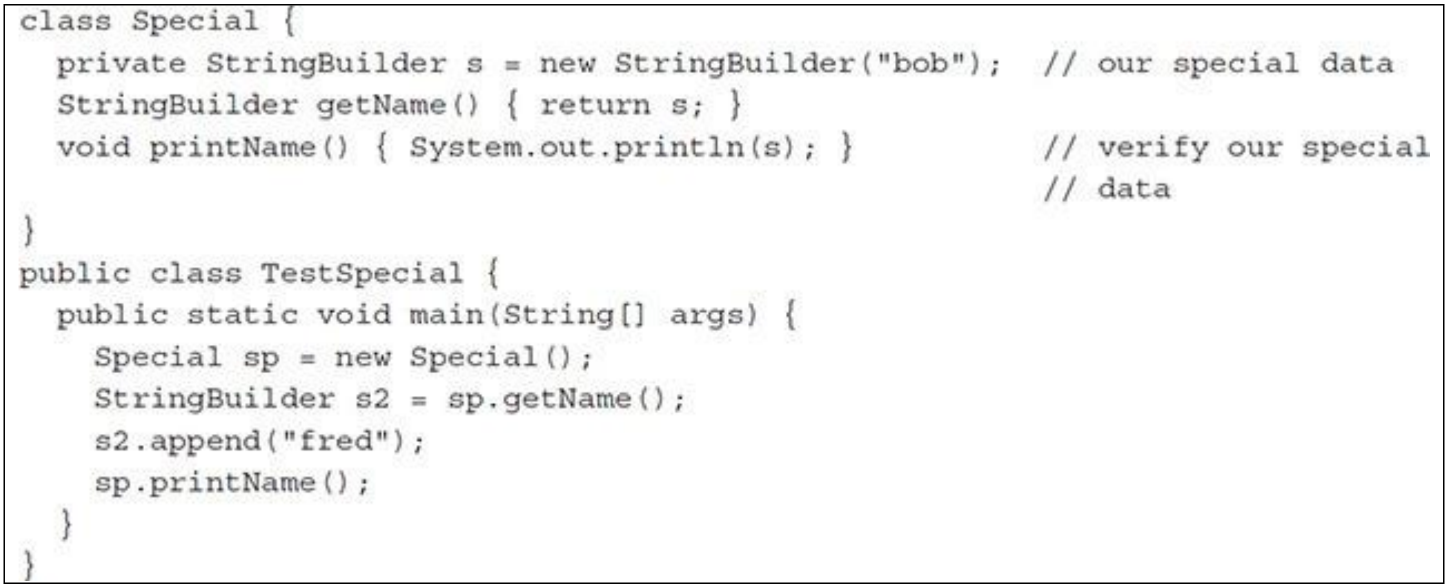
```
List<Integer> lista2 = new ArrayList<Integer>();
List<Integer> lista2 = new ArrayList<>();
```

>>

5. Advanced Encapsulation (OCA Objective 6.5)

If you want to encapsulate mutable objects like `StringBuilder`s or arrays or `ArrayList`s, you cannot return a reference to these objects; you must first make a copy of the object and return a reference to the copy.

Any class that has a method that returns a reference to a mutable object is breaking encapsulation.



```
class Special {
  private StringBuilder s = new StringBuilder("bob");  // our special data
  StringBuilder getName() { return s; }
  void printName() { System.out.println(s); }          // verify our special
                                                       // data
}
public class TestSpecial {
  public static void main(String[] args) {
    Special sp = new Special();
    StringBuilder s2 = sp.getName();
    s2.append("fred");
    sp.printName();
  }
}
```

>>

6. Using Simple Lambdas (OCA Objective 9.5)

Lambdas allow you to pass bits of code from one method to another.
And the receiving method can run whatever complying code it is sent.

While there are many types of lambdas that Java 8 supports, for this exam, the only lambda type you need to know is the `Predicate`.

The `Predicate` interface has a single method to implement that's called `test()`, and it takes one argument and returns a `boolean`.

As the `Predicate.test()` method returns a boolean, it can be placed (mostly?) wherever a boolean expression can go, e.g., in `if`, `while`, `do`, and ternary statements.

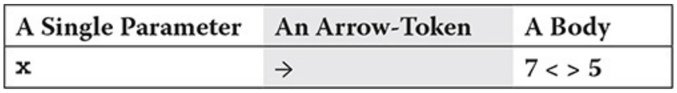
`Predicate` lambda expressions have three parts: a single argument, an arrow (`->`), and an expression or code block.

A `Predicate` lambda expression's argument can be just a variable or a type and variable together in parentheses, e.g., `(MyClass m)`.

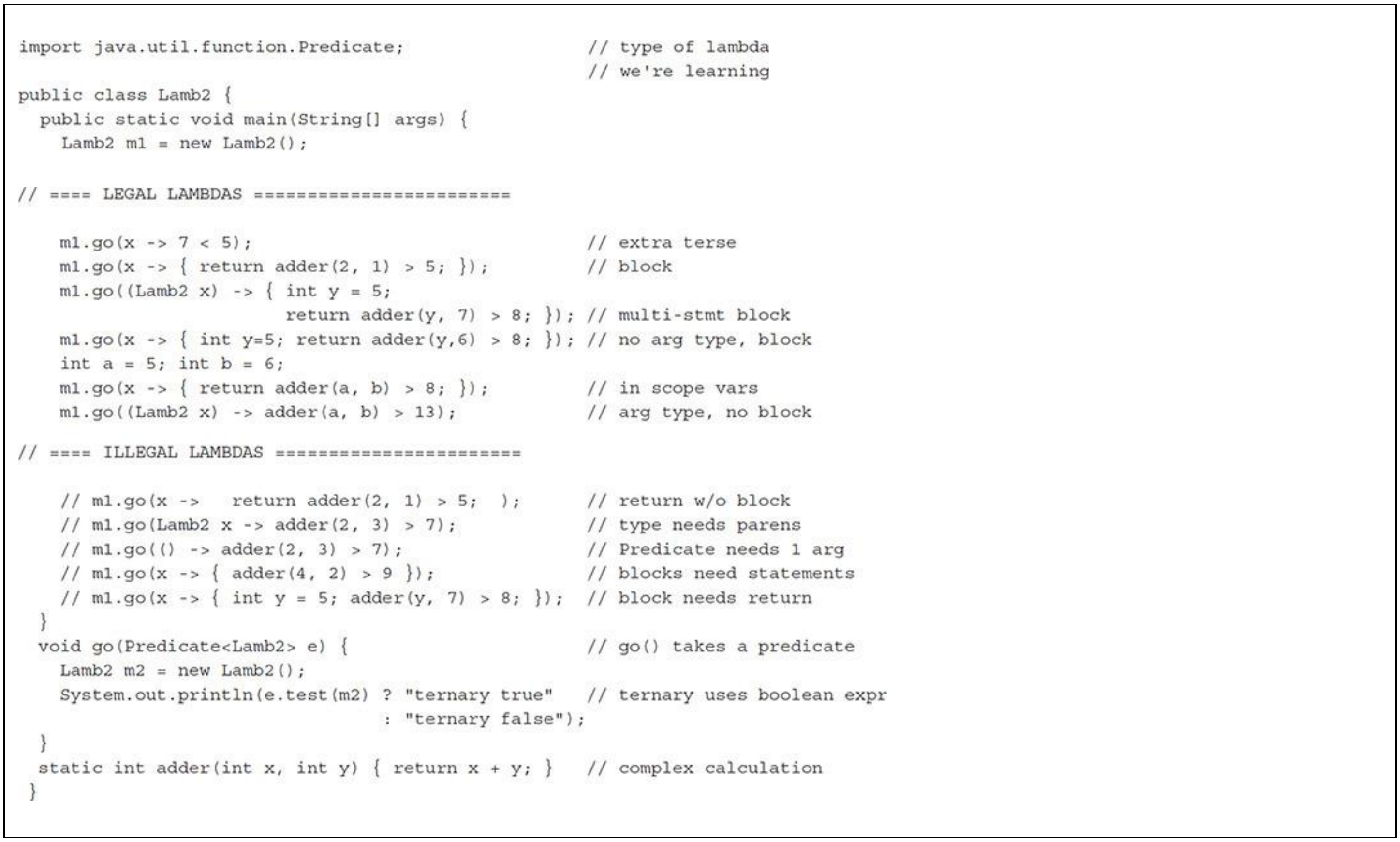
A `Predicate` lambda expression's body can be an expression that resolves to a boolean, OR it can be a block of statements (surrounded by curly braces) that ends with a boolean-returning `return` statement.

Java 8 is probably best known as the version of Java that finally added lambdas and streams.

The basic syntax for a `Predicate` lambda has three parts:



A Single Parameter	An Arrow-Token	A Body
x	→	7 < > 5



```java
import java.util.function.Predicate;                 // type of lambda
                                                     // we're learning
public class Lamb2 {
  public static void main(String[] args) {
    Lamb2 m1 = new Lamb2();

// ==== LEGAL LAMBDAS ========================

    m1.go(x -> 7 < 5);                               // extra terse
    m1.go(x -> { return adder(2, 1) > 5; });         // block
    m1.go((Lamb2 x) -> { int y = 5;
                         return adder(y, 7) > 8; }); // multi-stmt block
    m1.go(x -> { int y=5; return adder(y,6) > 8; }); // no arg type, block
    int a = 5; int b = 6;
    m1.go(x -> { return adder(a, b) > 8; });         // in scope vars
    m1.go((Lamb2 x) -> adder(a, b) > 13);            // arg type, no block

// ==== ILLEGAL LAMBDAS =======================

    // m1.go(x ->   return adder(2, 1) > 5;  );      // return w/o block
    // m1.go(Lamb2 x -> adder(2, 3) > 7);            // type needs parens
    // m1.go(() -> adder(2, 3) > 7);                 // Predicate needs 1 arg
    // m1.go(x -> { adder(4, 2) > 9 });              // blocks need statements
    // m1.go(x -> { int y = 5; adder(y, 7) > 8; });  // block needs return
  }
  void go(Predicate<Lamb2> e) {                      // go() takes a predicate
    Lamb2 m2 = new Lamb2();
    System.out.println(e.test(m2) ? "ternary true"   // ternary uses boolean expr
                                  : "ternary false");
  }
  static int adder(int x, int y) { return x + y; }   // complex calculation
 }
```

>>