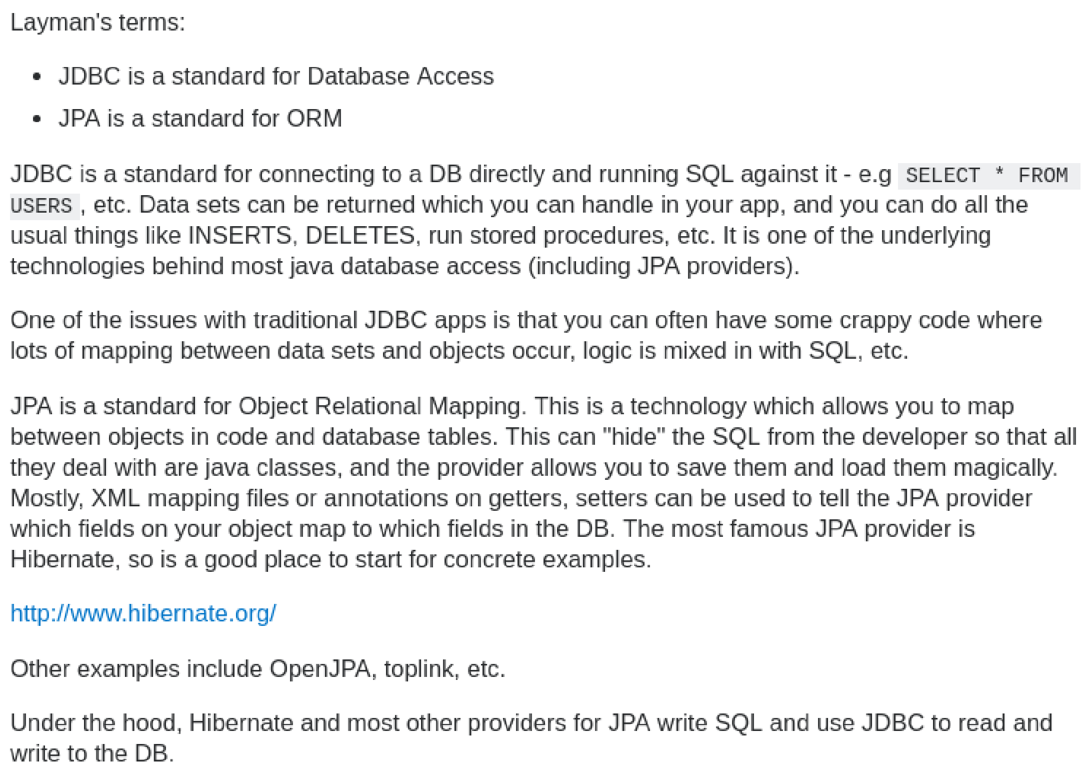
The difference between JPA and JDBC



Layman's terms:

- JDBC is a standard for Database Access
- JPA is a standard for ORM

JDBC is a standard for connecting to a DB directly and running SQL against it - e.g `SELECT * FROM USERS`, etc. Data sets can be returned which you can handle in your app, and you can do all the usual things like INSERTS, DELETES, run stored procedures, etc. It is one of the underlying technologies behind most java database access (including JPA providers).

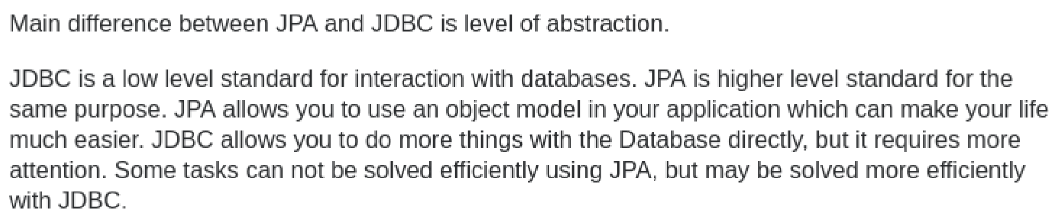
One of the issues with traditional JDBC apps is that you can often have some crappy code where lots of mapping between data sets and objects occur, logic is mixed in with SQL, etc.

JPA is a standard for Object Relational Mapping. This is a technology which allows you to map between objects in code and database tables. This can "hide" the SQL from the developer so that all they deal with are java classes, and the provider allows you to save them and load them magically. Mostly, XML mapping files or annotations on getters, setters can be used to tell the JPA provider which fields on your object map to which fields in the DB. The most famous JPA provider is Hibernate, so is a good place to start for concrete examples.

http://www.hibernate.org/

Other examples include OpenJPA, toplink, etc.

Under the hood, Hibernate and most other providers for JPA write SQL and use JDBC to read and write to the DB.



Main difference between JPA and JDBC is level of abstraction.

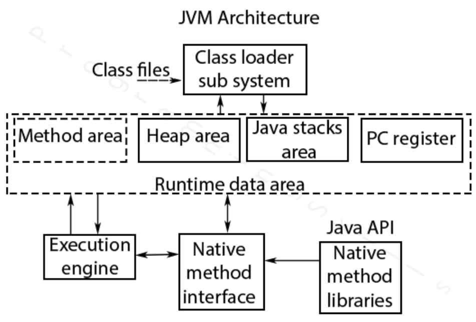
JDBC is a low level standard for interaction with databases. JPA is higher level standard for the same purpose. JPA allows you to use an object model in your application which can make your life much easier. JDBC allows you to do more things with the Database directly, but it requires more attention. Some tasks can not be solved efficiently using JPA, but may be solved more efficiently with JDBC.

→ JDBC is already implemented by Java. JPA is a standard/specification which different providers implement.

Differences between JDK, JRE and JVM

JVM:

1. Class Loader Subsystem



Java's dynamic class loading functionality is handled by the class loader subsystem. It loads, links. and initializes the class file when it refers to a class for the first time at runtime, not compile time.

1.1 Loading

Classes will be loaded by this component. Boot Strap class Loader, Extension class Loader, and Application class Loader are the three class loader which will help in achieving it.

- **Boot Strap ClassLoader** – Responsible for loading classes from the bootstrap classpath, nothing but rt.jar. Highest priority will be given to this loader.
- **Extension ClassLoader** – Responsible for loading classes which are inside ext folder (jre\lib).
- **Application ClassLoader** –Responsible for loading Application Level Classpath, path mentioned Environment Variable etc.

The above Class Loaders will follow Delegation Hierarchy Algorithm while loading the class files.

1.2 Linking

- **Verify** – Bytecode verifier will verify whether the generated bytecode is proper or not if verification fails we will get the verification error.
- **Prepare** – For all static variables memory will be allocated and assigned with default values.
- **Resolve** – All symbolic memory references are replaced with the original references from Method Area.

1.3 Initialization

This is the final phase of Class Loading, here all static variables will be assigned with the original values, and the static block will be executed.

2. Runtime Data Area

The Runtime Data Area is divided into 5 major components:

- **Method Area** – All the class level data will be stored here, including static variables. There is only one method area per JVM, and it is a shared resource.
- **Heap Area** – All the Objects and their corresponding instance variables and arrays will be stored here. There is also one Heap Area per JVM. Since the Method and Heap areas share memory for multiple threads, the data stored is not thread safe.
- **Stack Area** – For every thread, a separate runtime stack will be created. For every method call, one entry will be made in the stack memory which is called as Stack Frame. All local variables will be created in the stack memory. The stack area is thread safe since it is not a shared resource. The Stack Frame is divided into three sub-entities:
  - Local Variable Array – Related to the method how many local variables are involved and the corresponding values will be stored here.
  - Operand stack – If any intermediate operation is required to perform, operand stack acts as runtime workspace to perform the operation.
  - Frame data – All symbols corresponding to the method is stored here. In the case of any exception, the catch block information will be maintained in the frame data.
- **PC Registers** – Each thread will have separate PC Registers, to hold the address of current executing instruction once the instruction is executed the PC register will be updated with the next instruction.
- **Native Method stacks** – Native Method Stack holds native method information. For every thread, a separate native method stack will be created.

3. Execution Engine

The bytecode which is assigned to the Runtime Data Area will be executed by the Execution Engine. The Execution Engine reads the bytecode and executes it piece by piece.

- **Interpreter** – The interpreter interprets the bytecode faster, but executes slowly. The disadvantage of the interpreter is that when one method is called multiple times, every time a new interpretation is required.
- **JIT Compiler** – The JIT Compiler neutralizes the disadvantage of the interpreter. The Execution Engine will be using the help of the interpreter in converting byte code, but when it finds repeated code it uses the JIT compiler, which compiles the entire bytecode and changes it to native code. This native code will be used directly for repeated method calls, which improve the performance of the system.
    - Intermediate Code generator – Produces intermediate code
    - Code Optimizer – Responsible for optimizing the intermediate code generated above
    - Target Code Generator – Responsible for Generating Machine Code or Native Code
    - Profiler – A special component, responsible for finding hotspots, i.e. whether the method is called multiple times or not.
- **Garbage Collector** - Collects and removes unreferenced objects. Garbage Collection can be triggered by calling "**System.gc()**", but the execution is not guaranteed. Garbage collection of the JVM collects the objects that are created.

JIT Compiler

A JIT compiler runs **after** the program has started and compiles the code (usually bytecode or some kind of VM instructions) on the fly (or just-in-time, as it's called) into a form that's usually faster, typically the host CPU's native instruction set. A JIT has access to dynamic runtime information whereas a standard compiler doesn't and can make better optimizations like inlining functions that are used frequently.

This is in contrast to a traditional compiler that compiles **all** the code to machine language **before** the program is first run.

To paraphrase, conventional compilers build the whole program as an EXE file BEFORE the first time you run it. For newer style programs, an assembly is generated with pseudocode (p-code). Only AFTER you execute the program on the OS (e.g., by double-clicking on its icon) will the (JIT) compiler kick in and generate machine code (m-code) that the Intel-based processor or whatever will understand.

Typical scenario: The source code is completely converted into machine code

JIT scenario: The source code will be converted into assembly language like structure [for ex IL (intermediate language) for C#, ByteCode for java].

The intermediate code is converted into machine language only when the application needs that is required codes are only converted to machine code.

JIT vs Non-JIT comparison:

- In JIT not all the code is converted into machine code first a part of the code that is necessary will be converted into machine code then if a method or functionality called is not in machine then that will be turned into machine code... it reduces burden on the CPU.
- As the machine code will be generated on run time....the JIT compiler will produce machine code that is optimised for running machine's CPU architecture.

JIT Examples: In Java JIT is in JVM (Java Virtual Machine) In C# it is in CLR (Common Language Runtime) In Android it is in DVM (Dalvik Virtual Machine), or ART (Android RunTime) in newer versions.

A compiler would create machine language

No. A compiler is simply a program which takes as its input a program written in language **A** and produces as its output a semantically equivalent program in language **B**. Language **B** can be anything, it can be but doesn't have to be machine language.

which runs on the physical hardware directly?

Not necessarily. It could be run in an interpreter or in a VM. It could be further compiled to a different language.

I think you should drop the notion of "compiler versus interpreter" entirely, because it's a false dichotomy.

- A **compiler** is a transformer: It transforms a computer program written in a source language and outputs an equivalent in a target language. Usually, the source language is higher-level that the target language - and if it's the other way around, we often call that kind of transformer a decompiler.
- An **interpreter** is an execution engine. It executes a computer program written in one language, according to the specification of that language. We mostly use the term for software (but in a way, a classical CPU can be viewed as a hardware-based "interpreter" for its machine code).

Why does Java typically interpret instead of compile? The main advantage of compilation is that you end up with raw machine language code that can be efficiently executed on your machine. However, it can only be executed on one type of machine architecture (Intel Pentium, PowerPC). A primary advantage of a compiling to an intermediate language like Java bytecode and then interpreting is that you can achieve platform independence: you can interpret the same .class file on differently types of machine architectures. However, interpreting the bytecode is typically slower than executing pre-compiled machine language code. A second advantage of using the Java bytecode is that it acts as a buffer between your computer and the program. This enables you to download an untrusted program from the Internet and execute it on your machine with some assurances. Since you are running the Java interpreter (and not raw machine language code), you are protected by a layer of security which guards against malicious programs. It is the combination of Java and the Java bytecode that yield a platform-independent and secure environment, while still embracing a full set of modern programming abstractions.

The Java bytecode and the java interpreter are not inherently specific to the Java programming language. For example, you can use Jython to compile from the Python programming language into Java bytecode, and then use java to interpret it. There are similar ML, Lisp, and Fortran compilers that compile into JAva bytecode. You could also use the Unix program gcj to compile directly from a .java source file into a machine executable file a.out, which can be run natively on any Sparc microprocessors. Additionally, you can design hardware whose machine language is the Java bytecode. Sun Microsystems has done exactly this, making the Java Virtual Machine not so virtual.

Why not use a real machine language instead of the Java bytecode? The Java bytecode is much simpler than a typical high-level programming language. It is much easier to write a Java bytecode interpreter for a new type of computer than it is to write a full Java compiler. The abstraction means that it is much easier to reason about security and performance.

Why auto-import only `java.lang` package?

There is an implicit import of `java.lang.*`

A good reason not to autoimport too much is to avoid namespace clashes. If everything in java.util was imported automatically and then you wanted to refer to a different class named 'Map', for example, you would have to refer to it by its fully-qualified name.

Provides classes that are fundamental to the design of the Java programming language. The most important classes are `Object`, which is the root of the class hierarchy, and `Class`, instances of which represent classes at run time. `Wrapper classes`, `Math`, `Number`, `ProcessBuilder`, `Runtime` …

How does Java import work?

In Java, import is simply used by the compiler to let you name your classes by their unqualified name, let's say String instead of java.lang.String. You don't really need to import java.lang.* because the compiler does it by default. However this mechanism is just to save you some typing. Types in Java are fully qualified class names, so a String is really a java.lang.String object when the code is run. Packages are intended to prevent name clashes and allow two classes to have the same simple name, instead of relying on the old C convention of prefixing types like this. java_lang_String. This is called **namespacing**.
Hatırla, aynı paketde ise import'a gerek yok çünkü o classı görüyor **==** aynı namespace

BTW, in Java there's the **`static import`** construct, which allows to further save typing if you use lots of constants from a certain class. In a compilation unit (a .java file) which declares

```
import static java.lang.Math.*;
```

→ Classın içindeki bütün constantları ve methodlara direk erişebiliyorsun.

you can use the constant PI in your code, instead of referencing it through Math.PI, and the method cos() instead of Math.cos(). So for example you can write

```
double r = cos(PI * theta);
```

Java's import statement is pure syntactical sugar. import is only evaluated at **compile time** to indicate to the compiler where to find the names in the code.

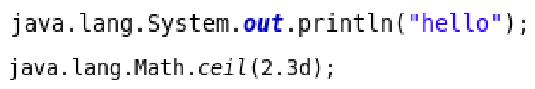
You may live without any import statement when you always specify the full qualified name of classes. Like this line needs no import statement at all:

```
javax.swing.JButton but = new javax.swing.JButton();
```

The import statement will make your code more readable like this:

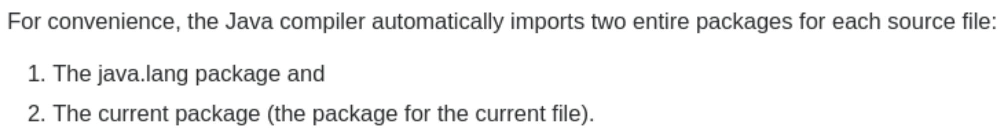
```
import javax.swing.*;
JButton but = new Jbutton();
```

- `java.lang` paketini auto-import edildiği için kısa versiyonunu yazıyoruz. Ancak bu şekilde de yazılabiliyor, aynısı başka paketler için de geçerli:



```
java.lang.System.out.println("hello");
java.lang.Math.ceil(2.3d);
```

- Ancak import ettiğimiz için, `Math` veya `Integer` veya `Double` yazdığımızda, ne olduğunu biliyor, bize "bu nedir bilmiyorum" diye bir hata vermiyor. Aynısı bütün paketler için geçerli. **Namespace** denilen olay bu.



For convenience, the Java compiler automatically imports two entire packages for each source file:

1. The java.lang package and
2. The current package (the package for the current file).

"Library Set" içinde neler var?

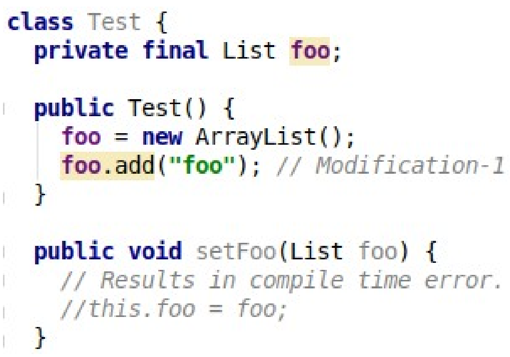
"Development tools" hangileri?

Tomcat javada yazılmış ve kendi VM i var ama tomcati hangi vm çalıştırıyor?

VM çalışınca instance mı oluşuyor? Bu durumda 2 VM instanceı mı oluyor 1 I tomcati çalıştıran, diğeri tomcatin içindeki servletları çalıştıran?

Final Keyword

This is one of the favorite interview questions. With this question, the interviewer tries to find out how well you understand the behavior of objects with respect to constructors, methods, class variables (static variables) and instance variables.



```
class Test {
  private final List foo;

  public Test() {
    foo = new ArrayList();
    foo.add("foo"); // Modification-1
  }

  public void setFoo(List foo) {
    // Results in compile time error.
    //this.foo = foo;
  }
```

In the above case, we have defined a constructor for 'Test' and gave it a 'setFoo' method.

About constructor: Constructor can be invoked only one time per object creation by using the new keyword. You cannot invoke constructor multiple times, because constructor are not designed to do so.

About method: A method can be invoked as many times as you want (Even never) and the compiler knows it.

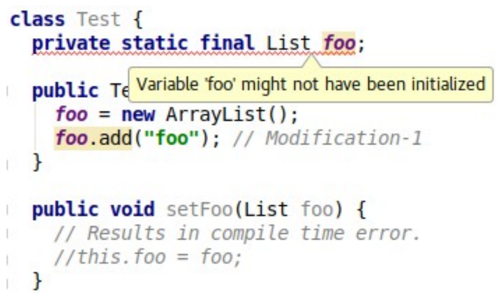
Scenario 1 `private final List foo;`

`foo` is an **instance** variable. When we create `Test` class object then the instance variable `foo`, will be copied inside the object of `Test` class. If we assign `foo` inside the constructor, then the compiler knows that the constructor will be invoked only once, so there is no problem assigning it inside the constructor.

If we assign `foo` inside a method, the compiler knows that a method can be called multiple times, which means the value will have to be changed multiple times, which is not allowed for a final variable. So the compiler decides constructor is good choice! **You can assign a value to a final variable only one time**.

Scenario 2 `private static final List foo = new ArrayList();`

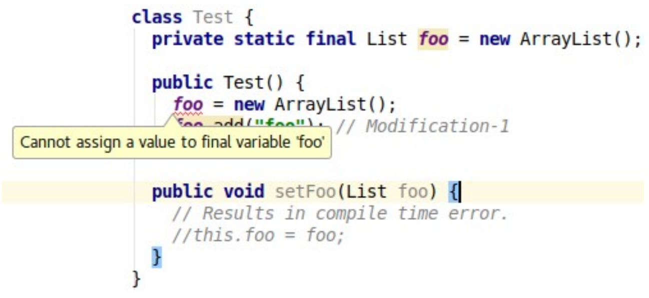
`foo` is now a **static** variable. When we create an instance of Test class, `foo` will not be copied to the object because `foo` is static. Now `foo` is not an independent property of each object. This is a property of Test class. But `foo` can be seen by multiple objects and if every object which is created by using the new keyword which will ultimately invoke the Test constructor which changes the value at the time of multiple object creation (Remember static `foo` is not copied in every object, but is shared between multiple objects.)



```
class Test {
  private static final List foo;

  public Te  Variable 'foo' might not have been initialized
    foo = new ArrayList();
    foo.add("foo"); // Modification-1
  }

  public void setFoo(List foo) {
    // Results in compile time error.
    //this.foo = foo;
  }
```



```
class Test {
  private static final List foo = new ArrayList();

  public Test() {
    foo = new ArrayList();
    foo.add("foo"); // Modification-1
Cannot assign a value to final variable 'foo'

  public void setFoo(List foo) {
    // Results in compile time error.
    //this.foo = foo;
  }
}
```

Scenario 3

Above Modification-2 is from your question. In the above case, you are not changing the first referenced object, but you are adding content inside foo which is allowed. Compiler complains if you try to assign a new ArrayList() to the foo reference variable. Rule If you have initialized a final variable, then you cannot change it to refer to a different object. (In this case ArrayList)

final classes cannot be subclassed
final methods cannot be overridden. (This method is in superclass)
final methods can override. (Read this in grammatical way. This method is in a subclass)

Final keyword has a numerous way to use:

- A final **class** cannot be subclassed.
- A final **method** cannot be overridden by subclasses
- A final **variable** can only be initialized once

Other usage:

- *When an anonymous inner class is defined within the body of a method, all variables declared final in the scope of that method are accessible from within the inner class*

A static class variable will exist from the start of the JVM, and should be initialized in the class. The error message won't appear if you do this.

The `final` keyword can be interpreted in two different ways depending on what it's used on:

Value types: For `int` s, `double` s etc, it will ensure that the value cannot change,

Reference types: For references to objects, `final` ensures that the **reference** will never change, meaning that it will always refer to the same object. It makes no guarantees whatsoever about the values inside the object being referred to staying the same.

As such, `final List<Whatever> foo;` ensures that `foo` always refers to *the same* list, but *the contents* of said list may change over time.

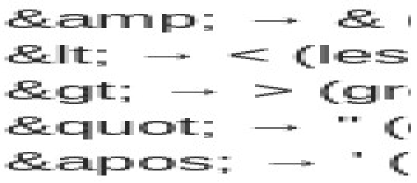
Hatırlatma:

- **URL escaping** ve **HTML escaping** farklı şeyler.

Mantıkları aynı. Programlama dillerinde de farklı karakterler escape ediliyor.

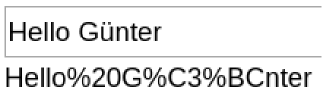
Örnek: Java'da **char escaping** var. ekrana **\t** yazdırmak istiyoruz ancak **print("\t")** yazarsak programlama dili onu bizim isteiğimiz gibi anlamayacak, o yüzden **print("\\t")** yazmak gerek. **\"** , **\'** ...

HTML escaping: ekrana <h3> yazdırmak istiyoruz ancak html parser onu "yanlış" anlayacak. O yüzden escape etmemiz gerek.



```
&amp;  →  & (
&lt;   →  < (les
&gt;   →  > (gr
&quot; →  " (
&apos; →  ' (
```

URL escaping / encoding (Percent Encoding): ASCII olmayan her şeyi ve boşlukları vs encode etmemiz lazım. UTF-8 hex karşılığı ile sanırım.



Hello Günter
Hello%20G%C3%BCnter

URLs can only be sent over the Internet using the ASCII character-set.

Since URLs often contain characters outside the ASCII set, the URL has to be converted into a valid ASCII format.

URL encoding replaces unsafe ASCII characters with a "%" followed by two hexadecimal digits.

URLs cannot contain spaces. URL encoding normally replaces a space with a plus (+) sign or with %20.

→ **ASCII'de ü** olmadığı için, ü'yü ascii tabelasındaki harflerle nasıl gösterebiliriz? **Ü**'nün **unicode karşılığını UTF-8 formatı** ile, yani hex değerleri ile gösteririz. % ve bütün hex değerleri ASCII var.

What is the difference between JSF, Servlet and JSP?

JSP (JavaServer Pages)

JSP is a **Java view technology** running on the server machine which allows you to write template text in client side languages (like HTML, CSS, JavaScript, ect.). JSP supports taglibs, which are backed by pieces of Java code that let you control the page flow or output dynamically. A well-known taglib is JSTL. JSP also supports Expression Language, which can be used to access backend data (via attributes available in the page, request, session and application scopes), mostly in combination with taglibs.

When a JSP is requested for the first time or when the web app starts up, the servlet container will compile it into a class extending `HttpServlet` and use it during the web app's lifetime. You can find the generated source code in the server's work directory. In for example Tomcat, it's the `/work` directory. On a JSP request, the servlet container will execute the compiled JSP class and send the generated output (usually just HTML/CSS/JS) through the web server over a network to the client side, which in turn displays it in the web browser.

Servlets

Servlet is a **Java application programming interface (API)** running on the server machine, which intercepts requests made by the client and generates/sends a response. A well-known example is the `HttpServlet` which provides methods to hook on HTTP requests using the popular HTTP methods such as `GET` and `POST`. You can configure `HttpServlet`s to listen to a certain HTTP URL pattern, which is configurable in `web.xml`, or more recently with Java EE 6, with `@WebServlet` annotation.

When a Servlet is first requested or during web app startup, the servlet container will create an instance of it and keep it in memory during the web app's lifetime. The same instance will be reused for every incoming request whose URL matches the servlet's URL pattern. You can access the request data by `HttpServletRequest` and handle the response by `HttpServletResponse`. Both objects are available as method arguments inside any of the overridden methods of `HttpServlet`, such as `doGet()` and `doPost()`.

JSF (JavaServer Faces)

JSF is a **component based MVC framework** which is built on top of the Servlet API and provides components via taglibs which can be used in JSP or any other Java based view technology such as Facelets. Facelets is much more suited to JSF than JSP. It namely provides great templating capabilities such as composite components, while JSP basically only offers the `<jsp:include>` for templating, so that you're forced to create custom components with raw Java code (which is a bit opaque and a lot of tedious work in JSF) when you want to replace a repeated group of components with a single component. Since JSF 2.0, JSP has been deprecated as view technology in favor of Facelets.

As being a MVC (Model-View-Controller) framework, JSF provides the `FacesServlet` as the sole request-response *Controller*. It takes all the standard and tedious HTTP request/response work from your hands, such as gathering user input, validating/converting them, putting them in model objects, invoking actions and rendering the response. This way you end up with basically a JSP or Facelets (XHTML) page for *View* and a JavaBean class as *Model*. The JSF components are used to bind the view with the model (such as your ASP.NET web control does) and the `FacesServlet` uses the *JSF component tree* to do all the work.

`==` checks for reference equality, however when writing code like:

```
Integer a = 1;
Integer b = 1;
```

Java is smart enough to reuse the same immutable for `a` and `b`, so this is true: `a == b`.

Is there one JVM per Java application?

Generally speaking, each application will get its own JVM instance and its own OS-level process and each JVM instance is independent of each other.

There are some implementation details such as Class Data Sharing, where multiple JVM instances might share some data/memory but those have no user-visible effect to the applications (except for improved startup time, hopefully).

A common scenario however is a single application server (or "web server") such as Glassfish or Tomcat running multiple web applications. In this case, multiple web applications can share a JVM.

There's one JVM per Java application. There shouldn't be any connection between them unless you establish one, e.g. with networking. If you're working inside of an IDE, the code you write generally runs in a separate JVM. The IDE will typically connect the separate JVM for debugging. If you're dealing with multiple web applications they could share the same JVM if they're deployed to the same web container.

The `equals()` Method

The `equals()` method compares two objects for equality and returns true if they are equal. **The equals() method provided in the Object class uses the `identity operator (==)` to determine whether two objects are equal.** For primitive data types, this gives the correct result. For objects, however, it does not.

The `equals()` method provided by Object **tests whether the object references are equal**—that is, if the objects compared are the exact same object. To test whether two objects are equal in the sense of equivalency (containing the same information), you **must override the equals()** method. Here is an example of a Book class that overrides equals():

```
public class Book {
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

`test1.equals(test2)` won't work as you expect if you don't override it.

Consider this code that tests two instances of the Book class for equality:

```
// Swing Tutorial, 2nd edition
Book firstBook  = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

This program displays objects are equal even though firstBook and secondBook reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number. You should always override the equals() method if the identity operator is not appropriate for your class.

Note: If you override `equals()`, you must override `hashCode()` as well.

You can't compare two `Integer` with a simple `==` they're objects so most of the time references won't be the same.

The issue is that your two Integer objects are just that, objects. They do not match because you are comparing your two object references, not the values within. Obviously `.equals` is overridden to provide a value comparison as opposed to an object reference comparison.

Objects that are equal (according to their *equals()*) must return the same hash code. **It's not required for different objects to return different hash codes.**

```
* <li>If two objects are equal according to the {@code equals(Object)}
*     method, then calling the {@code hashCode} method on each of
*     the two objects must produce the same integer result.
* <li>It is <em>not</em> required that if two objects are unequal
*     according to the {@link java.lang.Object#equals(java.lang.Object)}
*     method, then calling the {@code hashCode} method on each of the
*     two objects must produce distinct integer results.  However, the
*     programmer should be aware that producing distinct integer results
*     for unequal objects may improve the performance of hash tables.
```

Default implementation of equals() class provided by java.lang.Object **compares memory location and only** return true if two reference variable are pointing to same memory location i.e. essentially they are same object.

Java recommends to **override equals** and hashCode method if equality is going **to be defined by logical way or via some business logic**: example:

many classes in Java standard library does override it e.g. String overrides equals, whose implementation of equals() method return true if content of two String objects are exactly same

Integer wrapper class overrides equals to perform numerical comparison etc.

Don't forget to check the parameter type

```java
public class Foo {
  // some code

  public void equals(Object o) {
    Foo other = (Foo) o;
    // the real equals code
  }
}
```

In this example you are assuming something about the argument of equals(): You are assuming it's of type Foo. This needs not be the case! You can also get a String (in which case you should almost definitely return false).

So your code should look like this:

```java
public void equals(Object o) {
  if (!(o instanceof Foo)) {
    return false;
  }
  Foo other = (Foo) o;
  // the real equals code
}
```

```java
@Override
public boolean equals(Object obj) {

  // !obj.getClass().getName().equals("LinkedList.MyClass")
  // !(obj instanceof MyClass)
  // this.getClass() != obj.getClass()
  if (!(obj instanceof MyClass)){
    return false;
  }
  if (this.myField.equals(((MyClass) obj).getField())) {
    return true;
  }
  return false;
}
```

```java
public static void main(String[] args) {
  MyClass t1 = new MyClass( field: 16);
  MyClass t2 = new MyClass( field: 16);

  System.out.println(t1 == t2);

  System.out.println(t1.equals(t2));
```

```
false
true
```

The hashCode() Method

The value returned by hashCode() is the object's hash code, **which is the object's memory address in hexadecimal**. (Object class implementation)

By definition, if two objects are equal, their hash code must also be equal. If you override the equals() method, you change the way two objects are equated and Object's implementation of hashCode() is no longer valid. Therefore, if you override the equals() method, you must also override the hashCode() method as well.

```java
public static void main(String[] args) {
  MyClass t1 = new MyClass( field: 16);
  MyClass t2 = new MyClass( field: 16);

  System.out.println(t1 == t2);
  System.out.println(t1.equals(t2));
  System.out.println( t1.hashCode() == t2.hashCode());
```

```
false
true
false
```

```java
@Override
public int hashCode() {
  return Objects.hashCode(this.myField);
}

public static void main(String[] args) {
  MyClass t1 = new MyClass( field: 16);
  MyClass t2 = new MyClass( field: 16);

  System.out.println(t1 == t2);
  System.out.println(t1.equals(t2));
  System.out.println( t1.hashCode() == t2.hashCode());
```

```
false
true
true
```

As we know, if two objects are equal then their hashCode must also be equal. In the first example we did not override the hashCode() method, we only provided an equals(). We see that .equals() and hashCode() resulted in different booleans.

The second example shows how it should be.

```
public final class Objects
extends Object
```

Since:
1.7

This class consists of static utility methods for operating on objects. These utilities include null-safe or null-tolerant methods for computing the hash code of an object, returning a string for an object, and comparing two objects.

`compare(T a, T b, Comparator<? super T> c)`
Returns 0 if the arguments are identical and `c.compare(a, b)` otherwise.

`deepEquals(Object a, Object b)`
Returns `true` if the arguments are deeply equal to each other and `false` otherwise.

`equals(Object a, Object b)`
Returns `true` if the arguments are equal to each other and `false` otherwise.

`hash(Object... values)`
Generates a hash code for a sequence of input values.

`hashCode(Object o)`
Returns the hash code of a non-`null` argument and 0 for a `null` argument.

`requireNonNull(T obj)`
Checks that the specified object reference is not `null`.

`requireNonNull(T obj, String message)`
Checks that the specified object reference is not `null` and throws a customized `NullPointerException` if it is.

`toString(Object o)`
Returns the result of calling `toString` for a non-`null` argument and `"null"` for a `null` argument.

`toString(Object o, String nullDefault)`
Returns the result of calling `toString` on the first argument if the first argument is not `null` and returns the second argument otherwise.

Understanding How hashCode() Works

Simply put, hashCode() returns an integer value, generated by a hashing algorithm.

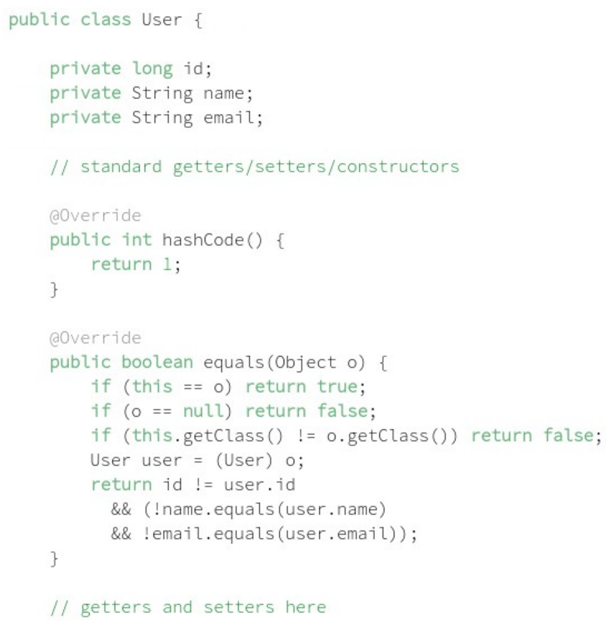
Objects that are equal (according to their equals()) must return the same hash code. **It's not required for different objects to return different hash codes**.

The general contract of hashCode() states:

- Whenever it is invoked on the same object more than once during an execution of a Java application, hashCode() must consistently return the same value, provided no information used in equals comparisons on the object is modified. This value needs not remain consistent from one execution of an application to another execution of the same application
- If two objects are equal according to the equals(Object) method, then calling the hashCode() method on each of the two objects must produce the same value
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, developers should be aware that producing distinct integer results for unequal objects improves the performance of hash tables

A Naive hashCode() Implementation

It's actually quite straightforward to have a naive hashCode() implementation that fully adheres to the above contract. To demonstrate this, we're going to define a sample User class that overrides the method's default implementation:



```java
public class User {

    private long id;
    private String name;
    private String email;

    // standard getters/setters/constructors

    @Override
    public int hashCode() {
        return 1;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null) return false;
        if (this.getClass() != o.getClass()) return false;
        User user = (User) o;
        return id != user.id
          && (!name.equals(user.name)
          && !email.equals(user.email));
    }

    // getters and setters here
```

The User class provides custom implementations for both **equals()** and **hashCode()** that fully adhere to the respective contracts. Even more, there's nothing illegitimate with having hashCode() returning any fixed value.

However, this implementation degrades the functionality of hash tables to basically zero, as every object would be stored in the same, single bucket.

In this context, a hash table lookup is performed linearly and does not give us any real advantage – more on this in section 7.

While it's essential to understand the roles that hashCode() and equals() methods play, we don't have to implement them from scratch every time, as most IDEs can generate custom hashCode() and equals() implementations and since **Java 7**, we got an **Objects.hash()** utility method for comfortable hashing.

- **An object's hashCode method must take the same fields into account as its equals method**.

Handling Hash Collisions

The intrinsic behavior of hash tables raises up a relevant aspect of these data structures: even with an efficient hashing algorithm, two or more objects might have the same hash code, even if they're unequal. So, their hash codes would point to the same bucket, even though they would have different hash table keys.

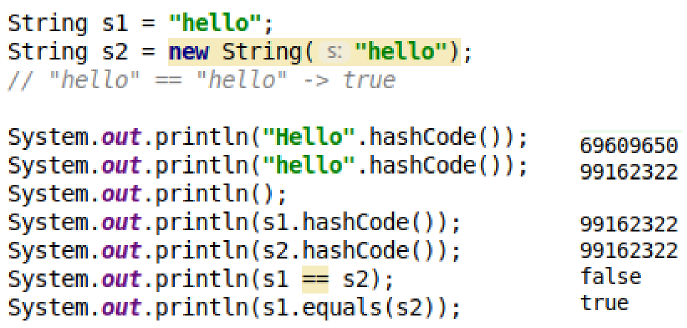
This situation is commonly known as a hash collision, and various methodologies exist for handling it, with each one having their pros and cons. Java's HashMap uses the separate chaining method for handling collisions:

"When two or more objects point to the same bucket, they're simply stored in a linked list. In such a case, the hash table is an array of linked lists, and each object with the same hash is appended to the linked list at the bucket index in the array.

In the worst case, several buckets would have a linked list bound to it, and the retrieval of an object in the list would be performed linearly."

Hash collision methodologies show in a nutshell why it's so important to implement hashCode() efficiently.

Java 8 brought an interesting enhancement to HashMap implementation – if a bucket size goes beyond the certain threshold, the linked list gets replaced with a tree map. This allows achieving O(logn) look up instead of pessimistic O(n).



```
String s1 = "hello";
String s2 = new String( s: "hello");
// "hello" == "hello" -> true

System.out.println("Hello".hashCode());     69609650
System.out.println("hello".hashCode());     99162322
System.out.println();
System.out.println(s1.hashCode());          99162322
System.out.println(s2.hashCode());          99162322
System.out.println(s1 == s2);               false
System.out.println(s1.equals(s2));          true
```

Where exactly does "business logic" lie in the MVC pattern?

The term business logic is in my opinion not a precise definition. Evans talks in his book, Domain Driven Design, about two types of business logic:

- Domain logic.
- Application logic.

This separation is in my opinion a lot clearer. And with the realization that there are different types of business rules also comes the realization that they don't all necessarily go the same place.

Domain logic is logic that corresponds to the actual domain. So if you are creating an accounting application, then domain rules would be rules regarding accounts, postings, taxation, etc. In an agile software planning tool, the rules would be stuff like calculating release dates based on velocity and story points in the backlog, etc.

For both these types of application, CSV import/export could be relevant, but the rules of CSV import/export has nothing to do with the actual domain. This kind of logic is application logic.

Domain logic most certainly goes into the model layer. The model would also correspond to the domain layer in DDD.

Application logic however does not necessarily have to be placed in the model layer. That could be placed in the controllers directly, or you could create a separate application layer hosting those rules. What is most logical in this case would depend on the actual application.

The **business logic** should be placed in the **model**, and we should be aiming for fat *models* and skinny *controllers*.

As a start point, we should start from the controller logic. For example: **on update**, your controller should direct your code to the *method/service* that **delivers** your changes to the model.

In the model, we may easily create *helper/service* classes where the application **business rules or calculations** can be validated.

A conceptual summary

- The controller is for application logic. The logic which is specific to how your application wants to interact with the "domain of knowledge" it belongs.
- The **model is for logic that is independent of the application**. This logic should be valid in all possible applications of the "domain of knowledge" it belongs.
- Thus, it is logical to place all business rules in the model.

ElYusubov's answer mostly nails it, domain logic should go into the model and application logic into the controller.

Two clarifications:

- The term business logic is rather useless here, because it is ambiguous. Business logic is an umbrella term for all logic that business-people care about, separating it from mere technicalities like how to store stuff in a database or how to render it on a screen. Both domain logic ("a valid email address looks like...") and workflows/business processes ("when a user signs up, ask for his/her email address") are considered business logic, with the former clearly belonging in the model and the latter being application logic that goes in the controller.
- MVC is a pattern for putting stuff on a screen and allowing the user to interact with it, it does not specify storage *at all*. Most MVC-frameworks are full stack frameworks that go beyond mere MVC and do help you with storing your data, and because the data that should be stored are usually to be found in the model, these frameworks give you convenient ways of storing your model-data in a database, but that has nothing to do with MVC. Ideally, models should be persistence-agnostic and switching to a different type of storage should not affect model-code at all. Full fledged architectures have a persistence layer to handle this.

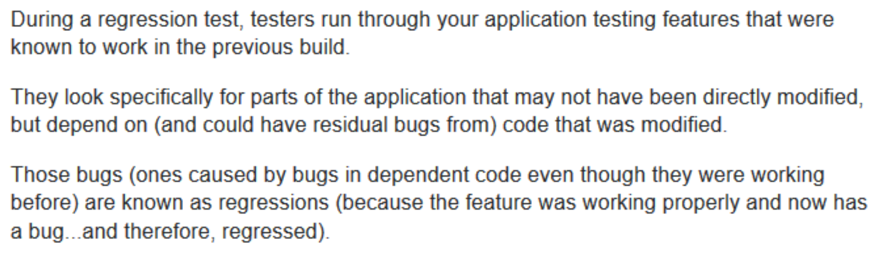
Testing

Regression Test

Regression test is a test that is performed to make sure that previously working functionality still works, after changes elsewhere in the system. Your unit tests are automatically regression tests, and that's one of their biggest advantages. Once those tests are written, they will be run in future, whenever you add new functionality or change existing functionality. You don't need to explicitly write regression tests.

- The intent of regression testing is to provide a general assurance that no additional errors were introduced in the process of fixing other problems.
- -1 This definition is extremely broad. Every test makes sure working functionality still works - that's the general point of testing. Regression tests are tests written during debugging to make sure the code does not regress. “A test that was written when a bug was fixed. It ensure that this specific bug will not occur again. The full name is "non-regression test".

Notwithstanding the old joke, "Congress" is not the opposite of "progress;" "regress" is. For your code to regress is for it to "move backward," typically meaning that some bad behavior it once had, which you fixed, has come back. A "regression" is the return of a bug (although there can be other interpretations). A regression test, therefore, is a test that validates that you have fixed the bug, and one that you run periodically to ensure that your fix is still in place, still working.

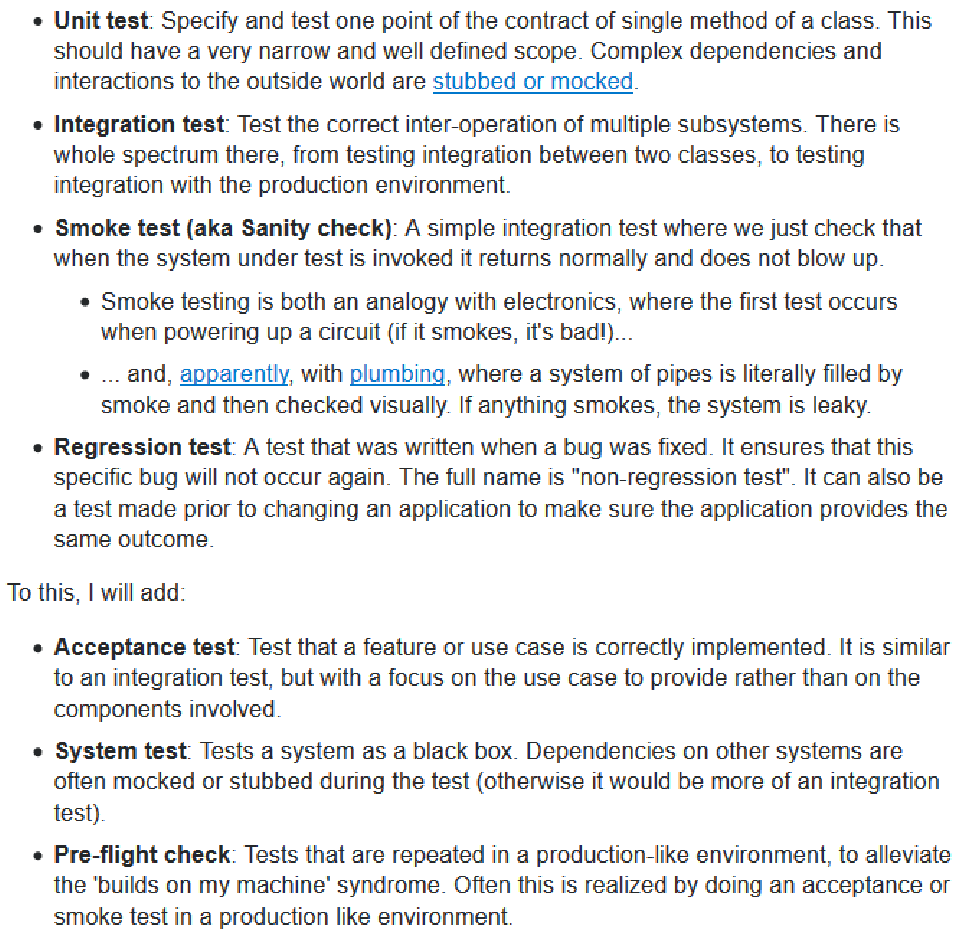


During a regression test, testers run through your application testing features that were known to work in the previous build.

They look specifically for parts of the application that may not have been directly modified, but depend on (and could have residual bugs from) code that was modified.

Those bugs (ones caused by bugs in dependent code even though they were working before) are known as regressions (because the feature was working properly and now has a bug...and therefore, regressed).

Other Kinds of Tests



- **Unit test**: Specify and test one point of the contract of single method of a class. This should have a very narrow and well defined scope. Complex dependencies and interactions to the outside world are stubbed or mocked.
- **Integration test**: Test the correct inter-operation of multiple subsystems. There is whole spectrum there, from testing integration between two classes, to testing integration with the production environment.
- **Smoke test (aka Sanity check)**: A simple integration test where we just check that when the system under test is invoked it returns normally and does not blow up.
  - Smoke testing is both an analogy with electronics, where the first test occurs when powering up a circuit (if it smokes, it's bad!)...
  - ... and, apparently, with plumbing, where a system of pipes is literally filled by smoke and then checked visually. If anything smokes, the system is leaky.
- **Regression test**: A test that was written when a bug was fixed. It ensures that this specific bug will not occur again. The full name is "non-regression test". It can also be a test made prior to changing an application to make sure the application provides the same outcome.

To this, I will add:

- **Acceptance test**: Test that a feature or use case is correctly implemented. It is similar to an integration test, but with a focus on the use case to provide rather than on the components involved.
- **System test**: Tests a system as a black box. Dependencies on other systems are often mocked or stubbed during the test (otherwise it would be more of an integration test).
- **Pre-flight check**: Tests that are repeated in a production-like environment, to alleviate the 'builds on my machine' syndrome. Often this is realized by doing an acceptance or smoke test in a production like environment.

How do I copy an object in Java?

556

216

Consider the below code:

```
DummyBean dum = new DummyBean();
dum.setDummy("foo");
System.out.println(dum.getDummy()); // prints 'foo'

DummyBean dumtwo = dum;
System.out.println(dumtwo.getDummy()); // prints 'foo'

dum.setDummy("bar");
System.out.println(dumtwo.getDummy()); // prints 'bar' but it should print 'foo'
```

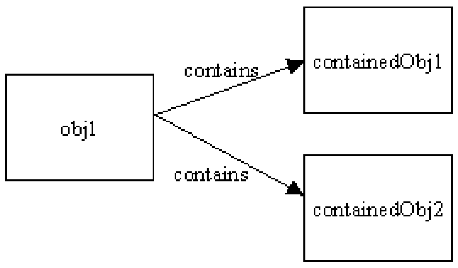
So, I want to copy the 'dum' to 'dumtwo' and I want to change 'dum' without affecting the 'dumtwo'. But the above code is not doing that. When I change something in 'dum', the same change is happening in 'dumtwo' also.

I guess, when I say `dumtwo = dum`, Java copies the **reference only**. So, is there any way to create a fresh copy of 'dum' and assign it to 'dumtwo'?

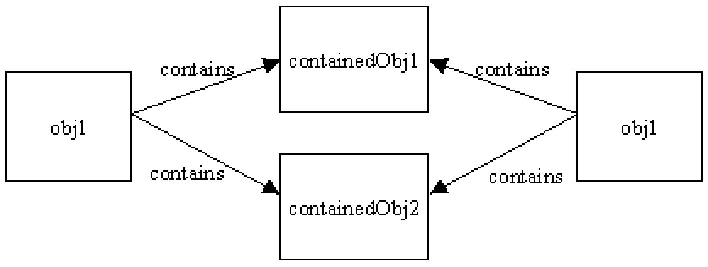
313

Basic: Object Copying in Java.

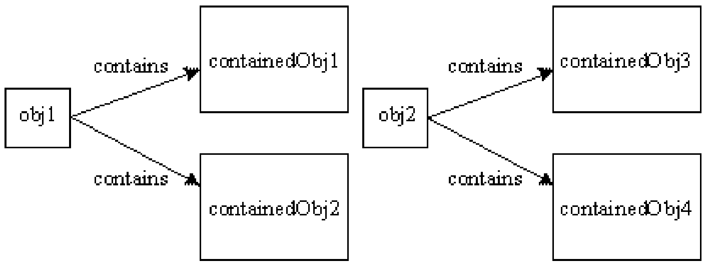
Let us Assume an object- `obj1`, that contains two objects, **containedObj1** and **containedObj2**.



shallow copying:
shallow copying creates a new `instance` of the same class and copies all the fields to the new instance and returns it. **Object class** provides a `clone` method and provides support for the shallow copying.



Deep copying:
A deep copy occurs when *an object is copied along with the objects to which it refers*. Below image shows `obj1` after a deep copy has been performed on it. *Not only has* `obj1` *been copied*, but the objects contained within it have been copied as well. We can use `Java Object Serialization` to make a deep copy. Unfortunately, this approach has some problems too(detailed examples).



Possible Problems:
`clone` is tricky to implement correctly.
It's better to use Defensive copying, copy constructors(as @egaga reply) or static factory methods.

1. If you have an object, that you know has a public `clone()` method, but you don't know the type of the object at compile time, then you have problem. Java has an interface called `Cloneable`. In practice, we should implement this interface if we want to make an object `Cloneable`. `Object.clone` is **protected**, so we must *override* it with a public method in order for it to be accessible.
2. Another problem arises when we try **deep copying** of a *complex object*. Assume that the `clone()` method of all member object variables also does deep copy, this is too risky of an assumption. You must control the code in all classes.

For example org.apache.commons.lang.SerializationUtils will have method for Deep clone using serialization(Source). If we need to clone Bean then there are couple of utility methods in org.apache.commons.beanutils (Source).

- `cloneBean` will Clone a bean based on the available property getters and setters, even if the bean class itself does not implement Cloneable.
- `copyProperties` will Copy property values from the origin bean to the destination bean for all cases where the property names are the same.

What is a JavaBean exactly? / Java Serialization

A JavaBean is just a standard

1. All properties private (use getters/setters)
2. A public no-argument constructor
3. Implements `Serializable`.

That's it. It's just a convention. Lots of libraries depend on it though.

With respect to `Serializable`, from the API documentation:

> Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.

In other words, serializable objects can be written to streams, and hence files, object databases, anything really.

Also, there is no syntactic difference between a JavaBean and another class -- a class is a JavaBean if it follows the standards.

There is a term for it because the standard allows libraries to programmatically do things with class instances you define in a predefined way. For example, if a library wants to stream any object you pass into it, it knows it can because your object is serializable (assuming the lib requires your objects be proper JavaBeans).

There's a term for it to make it sound special. The reality is nowhere near so mysterious.

Basically, a "Bean":

- is a serializable object (that is, it implements `java.io.Serializable`, and does so correctly), that
- has "properties" whose getters and setters are just methods with certain names (like, say, `getFoo()` is the getter for the "Foo" property), and
- has a public 0-arg constructor (so it can be created at will and configured by setting its properties).

Update:

As for `Serializable`: That is nothing but a "marker interface" (an interface that doesn't declare any functions) that tells Java that the implementing class consents to (and implies that it is capable of) "serialization" -- a process that converts an instance into a stream of bytes. Those bytes can be stored in files, sent over a network connection, etc, and have enough info to allow a JVM (at least, one that knows about the object's type) to reconstruct the object later -- possibly in a different instance of the application, or even on a whole other machine!

Of course, in order to do that, the class has to abide by certain limitations. Chief among them is that all instance fields must be either primitive types (int, bool, etc), instances of some class that is also serializable, or marked as `transient` so that Java won't try to include them. (This of course means that `transient` fields will not survive the trip over a stream. A class that has `transient` fields should be prepared to reinitialize them if necessary.)

A class that can not abide by those limitations should not implement `Serializable` (and, IIRC, the Java compiler won't even *let* it do so.)

Daring to answer 6 year old question, adding just a very high level understanding for people new to Java

> What is Serialization?

Converting an object to bytes and bytes back to object (Deserialization).

> when is serialization used?

When we want to Persist the Object. When we want the object to exist beyond the lifetime of the JVM.

> Real World Example:

ATM: When the account holder tries to withdraw money from the server through ATM, the account holder information like withdrawl details will be serialized and sent to server where the details are deserialized and used to perform operations.

> How serialization is performed in java.

1. Implement `java.io.Serializable` interface (marker interface so no method to implement).
2. Persist the object: Use `java.io.ObjectOutputStream` class, a filter stream which is a wrapper around a lower-level byte stream (to write Object to file systems or transfer a flattened object across a network wire and rebuilt on the other side).
   - `writeObject(<<instance>>)` - to write an object
   - `readObject()` - to read an serialized Object

> Remember:

When you serialize an object, only the object's state will be saved, not the object's class file or methods.

When you serialized a 2 byte object, you see 51 bytes serialized file.

> Steps how the object is serialized and de-serialized.

Answer for: How did it convert to 51 bytes file?

- First writes the serialization stream magic data (STREAM_MAGIC= "AC ED" and STREAM_VERSION=version of the JVM).
- Then it writes out the metadata of the class associated with an instance (length of the class, the name of the class, serialVersionUID).
- Then it recursively writes out the metadata of the superclass until it finds `java.lang.Object`.
- Then starts with the actual data associated with the instance.
- Finally writes the data of objects associated with the instance starting from metadata to actual content.

Edit : One more good link to read.

This will answer a few frequent questions:

1. How not to serialize any field in class.
   Ans: use transient keyword
2. When child class is serialized does parent class get serialized?
   Ans: No, If parent is not extending Serializable interface parents field don't get serialized.
3. When parent is serialized does child class get serialized?
   Ans: Yes, by default child class also get serialized.
4. How to avoid child class from getting serialized?
   Ans: a. Override writeObject and readObject method and throw `NotSerializableException`.

   b. also you can mark all fields transient in child class.
5. Some system-level classes such as Thread, OutputStream and its subclasses, and Socket are not serializable.

In computer science, in the context of data storage, serialization (or serialisation) is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer) or transmitted (for example, across a network connection link) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread **according to the serialization format**, it can be used to create a semantically identical clone of the original object. For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects **does not include** any of their associated **methods** with which they were previously linked.

This process of serializing an object is also called **marshalling** an object. The opposite operation, extracting a data structure from a series of bytes, is **deserialization** (also called **unmarshalling**).

Objects that implement `Serializable` will generally be serialized using native Java serialization algorithm (documented here).

Simple Example:

```
class Save implements Serializable
{
    int i;
}
```

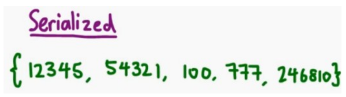
```
Save obj = new Save();
obj.i = 4;

File f = new File("Obj.txt");
FileOutputStream fos = new FileOutputStream(f);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(obj);
```

```
¬í |sr ┘Save
Å è-°Åú┐  I  ixp   ┘
```

➢ There are many libraries in Java that support serialization of Java objects to JSON and back. json-lib, Json-io, Flexjson, Jackson, Gson ...

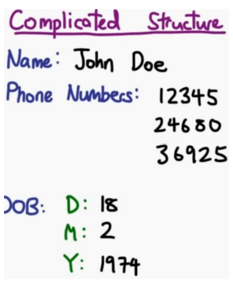
For simple data structures, for example an integer array might be serialized like this:



Serialized

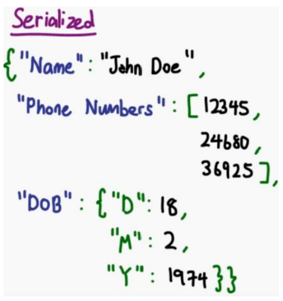
{12345, 54321, 100, 777, 246810}

Whereas a more complicated structure with nested arrays, nested objects etc might be converted to this:



Complicated Structure

Name: John Doe
Phone Numbers: 12345
24680
36925
DOB: D: 18
M: 2
Y: 1974



Serialized

```
{"Name": "John Doe",
 "Phone Numbers": [12345,
                   24680,
                   36925],
 "DoB": {"D": 18,
         "M": 2,
         "Y": 1974}}
```