

PROMETEO

Unidad 4: Funciones y Modularización Avanzada

Módulo Profesional Optativo I

Técnico Superior de DAM / DAW



Fundamentos

Comprender cómo se pasan los parámetros en Java es una de las claves para evitar errores lógicos en programas complejos. Aunque muchos lenguajes diferencian entre **paso por valor** y **paso por referencia**, Java adopta un modelo intermedio y, a menudo, malinterpretado: **todo se pasa por valor**, pero lo que se copia puede ser un valor primitivo o una **referencia** a un objeto.

Esto significa que el mecanismo de paso de argumentos depende del **tipo de dato**:

Tipos primitivos

(int, double, boolean, etc.): se pasa una **copia del valor**. Cualquier modificación dentro del método no afecta a la variable original.

```
void cambiar(int x) { x = 10; }
int a = 5;
cambiar(a); // 'a' sigue siendo 5
```

Tipos objeto

(clases, arreglos, etc.): se pasa una **copia de la referencia**, no del objeto en sí. Esto implica que si se modifican propiedades internas del objeto dentro del método, los cambios se reflejarán fuera. Sin embargo, si se **reasigna** la variable a un nuevo objeto, el original no cambia.

```
void modificar(Persona p) {
    p.nombre = "Ana"; // modifica el
    objeto original
}
Persona p1 = new Persona("Luis");
modificar(p1); // p1.nombre ahora es
"Ana"
```

En este ejemplo, la referencia (una dirección de memoria) se copia. Por tanto, **ambas variables apuntan al mismo objeto**. Sin embargo, si dentro del método hacemos `p = new Persona("Carla");`, solo cambiamos la copia local de la referencia; el objeto original sigue siendo el mismo.

Esta distinción es fundamental porque muchas veces los errores en programas Java surgen de suposiciones incorrectas sobre cómo se comportan las variables dentro de los métodos. Cuando un programador espera que un método "devuelva" modificaciones sobre una variable primitiva, no lo hará, a menos que se devuelva explícitamente un nuevo valor. En cambio, si el método modifica los atributos de un objeto, sí veremos cambios reflejados en el original.

□ Aplicación práctica

En la práctica profesional, esta diferencia se traduce en **decisiones de diseño**:

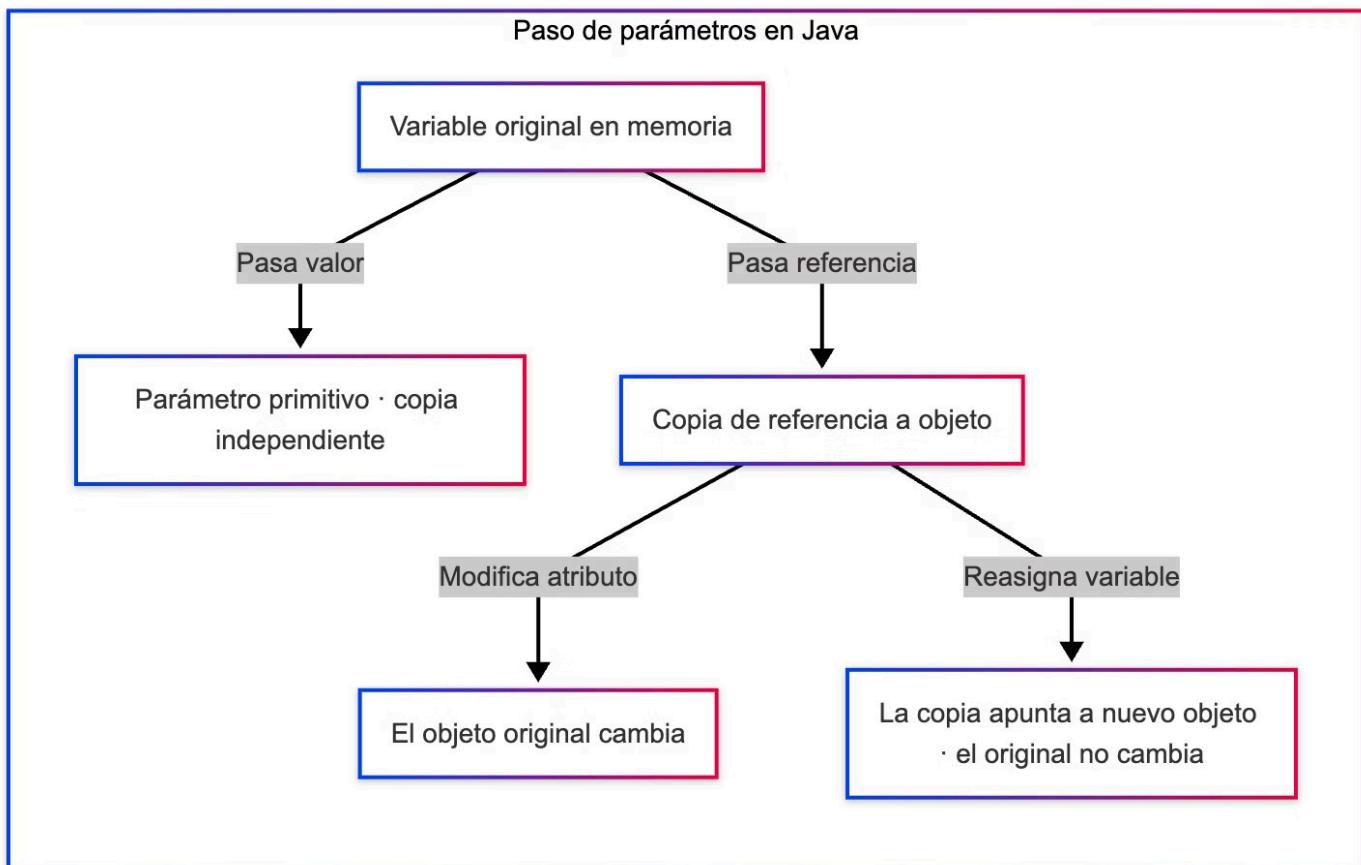
- Si queremos **proteger el estado de un objeto**, debemos crear copias (clonación).
- Si queremos **eficiencia**, trabajaremos sobre referencias, asumiendo los posibles efectos colaterales.
- En proyectos colaborativos o sistemas concurrentes (por ejemplo, servidores web), el mal uso de referencias puede provocar **inconsistencias de datos** si varios hilos modifican el mismo objeto compartido.

Por eso, muchas bibliotecas modernas (como las de fechas, LocalDate, LocalTime, etc.) implementan **objetos inmutables**, que no pueden modificarse una vez creados. Así se garantiza seguridad y consistencia en entornos multihilo.

```
attachEvent("onreadystatechange",H),e.attachEvent("onload",H);var b=window,document;var a={};function F(e){var t=_[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var _={};function G(e){var t=_[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var H={};function I(e){var t=H[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var J={};function K(e){var t=J[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var L={};function M(e){var t=L[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var N={};function O(e){var t=N[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var P={};function Q(e){var t=P[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var R={};function S(e){var t=R[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var T={};function U(e){var t=T[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var V={};function W(e){var t=V[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var X={};function Y(e){var t=X[e]={};return b.createElement(t).style.cssText=t,e.type=="checkbox"?t.checked==e.checked:t.value==e.value,t}var Z={};function $(_){var e=_[0];var t=_[1];var n=_[2];var r=_[3];var s=_[4];var i=_[5];var o=_[6];var u=_[7];var l=_[8];var c=_[9];var d=_[10];var h=_[11];var p=_[12];var f=_[13];var g=_[14];var m=_[15];var v=_[16];var y=_[17];var w=_[18];var k=_[19];var x=_[20];var z=_[21];var b=_[22];var j=_[23];var a=_[24];var q=_[25];var r=_[26];var s=_[27];var i=_[28];var o=_[29];var u=_[30];var l=_[31];var c=_[32];var d=_[33];var h=_[34];var p=_[35];var f=_[36];var g=_[37];var m=_[38];var v=_[39];var y=_[40];var w=_[41];var k=_[42];var x=_[43];var z=_[44];var b=_[45];var j=_[46];var a=_[47];var q=_[48];var r=_[49];var s=_[50];var i=_[51];var o=_[52];var u=_[53];var l=_[54];var c=_[55];var d=_[56];var h=_[57];var p=_[58];var f=_[59];var g=_[60];var m=_[61];var v=_[62];var y=_[63];var w=_[64];var k=_[65];var x=_[66];var z=_[67];var b=_[68];var j=_[69];var a=_[70];var q=_[71];var r=_[72];var s=_[73];var i=_[74];var o=_[75];var u=_[76];var l=_[77];var c=_[78];var d=_[79];var h=_[80];var p=_[81];var f=_[82];var g=_[83];var m=_[84];var v=_[85];var y=_[86];var w=_[87];var k=_[88];var x=_[89];var z=_[90];var b=_[91];var j=_[92];var a=_[93];var q=_[94];var r=_[95];var s=_[96];var i=_[97];var o=_[98];var u=_[99];var l=_[100];var c=_[101];var d=_[102];var h=_[103];var p=_[104];var f=_[105];var g=_[106];var m=_[107];var v=_[108];var y=_[109];var w=_[110];var k=_[111];var x=_[112];var z=_[113];var b=_[114];var j=_[115];var a=_[116];var q=_[117];var r=_[118];var s=_[119];var i=_[120];var o=_[121];var u=_[122];var l=_[123];var c=_[124];var d=_[125];var h=_[126];var p=_[127];var f=_[128];var g=_[129];var m=_[130];var v=_[131];var y=_[132];var w=_[133];var k=_[134];var x=_[135];var z=_[136];var b=_[137];var j=_[138];var a=_[139];var q=_[140];var r=_[141];var s=_[142];var i=_[143];var o=_[144];var u=_[145];var l=_[146];var c=_[147];var d=_[148];var h=_[149];var p=_[150];var f=_[151];var g=_[152];var m=_[153];var v=_[154];var y=_[155];var w=_[156];var k=_[157];var x=_[158];var z=_[159];var b=_[160];var j=_[161];var a=_[162];var q=_[163];var r=_[164];var s=_[165];var i=_[166];var o=_[167];var u=_[168];var l=_[169];var c=_[170];var d=_[171];var h=_[172];var p=_[173];var f=_[174];var g=_[175];var m=_[176];var v=_[177];var y=_[178];var w=_[179];var k=_[180];var x=_[181];var z=_[182];var b=_[183];var j=_[184];var a=_[185];var q=_[186];var r=_[187];var s=_[188];var i=_[189];var o=_[190];var u=_[191];var l=_[192];var c=_[193];var d=_[194];var h=_[195];var p=_[196];var f=_[197];var g=_[198];var m=_[199];var v=_[200];var y=_[201];var w=_[202];var k=_[203];var x=_[204];var z=_[205];var b=_[206];var j=_[207];var a=_[208];var q=_[209];var r=_[210];var s=_[211];var i=_[212];var o=_[213];var u=_[214];var l=_[215];var c=_[216];var d=_[217];var h=_[218];var p=_[219];var f=_[220];var g=_[221];var m=_[222];var v=_[223];var y=_[224];var w=_[225];var k=_[226];var x=_[227];var z=_[228];var b=_[229];var j=_[230];var a=_[231];var q=_[232];var r=_[233];var s=_[234];var i=_[235];var o=_[236];var u=_[237];var l=_[238];var c=_[239];var d=_[240];var h=_[241];var p=_[242];var f=_[243];var g=_[244];var m=_[245];var v=_[246];var y=_[247];var w=_[248];var k=_[249];var x=_[250];var z=_[251];var b=_[252];var j=_[253];var a=_[254];var q=_[255];var r=_[256];var s=_[257];var i=_[258];var o=_[259];var u=_[260];var l=_[261];var c=_[262];var d=_[263];var h=_[264];var p=_[265];var f=_[266];var g=_[267];var m=_[268];var v=_[269];var y=_[270];var w=_[271];var k=_[272];var x=_[273];var z=_[274];var b=_[275];var j=_[276];var a=_[277];var q=_[278];var r=_[279];var s=_[280];var i=_[281];var o=_[282];var u=_[283];var l=_[284];var c=_[285];var d=_[286];var h=_[287];var p=_[288];var f=_[289];var g=_[290];var m=_[291];var v=_[292];var y=_[293];var w=_[294];var k=_[295];var x=_[296];var z=_[297];var b=_[298];var j=_[299];var a=_[300];var q=_[301];var r=_[302];var s=_[303];var i=_[304];var o=_[305];var u=_[306];var l=_[307];var c=_[308];var d=_[309];var h=_[310];var p=_[311];var f=_[312];var g=_[313];var m=_[314];var v=_[315];var y=_[316];var w=_[317];var k=_[318];var x=_[319];var z=_[320];var b=_[321];var j=_[322];var a=_[323];var q=_[324];var r=_[325];var s=_[326];var i=_[327];var o=_[328];var u=_[329];var l=_[330];var c=_[331];var d=_[332];var h=_[333];var p=_[334];var f=_[335];var g=_[336];var m=_[337];var v=_[338];var y=_[339];var w=_[340];var k=_[341];var x=_[342];var z=_[343];var b=_[344];var j=_[345];var a=_[346];var q=_[347];var r=_[348];var s=_[349];var i=_[350];var o=_[351];var u=_[352];var l=_[353];var c=_[354];var d=_[355];var h=_[356];var p=_[357];var f=_[358];var g=_[359];var m=_[360];var v=_[361];var y=_[362];var w=_[363];var k=_[364];var x=_[365];var z=_[366];var b=_[367];var j=_[368];var a=_[369];var q=_[370];var r=_[371];var s=_[372];var i=_[373];var o=_[374];var u=_[375];var l=_[376];var c=_[377];var d=_[378];var h=_[379];var p=_[380];var f=_[381];var g=_[382];var m=_[383];var v=_[384];var y=_[385];var w=_[386];var k=_[387];var x=_[388];var z=_[389];var b=_[390];var j=_[391];var a=_[392];var q=_[393];var r=_[394];var s=_[395];var i=_[396];var o=_[397];var u=_[398];var l=_[399];var c=_[400];var d=_[401];var h=_[402];var p=_[403];var f=_[404];var g=_[405];var m=_[406];var v=_[407];var y=_[408];var w=_[409];var k=_[410];var x=_[411];var z=_[412];var b=_[413];var j=_[414];var a=_[415];var q=_[416];var r=_[417];var s=_[418];var i=_[419];var o=_[420];var u=_[421];var l=_[422];var c=_[423];var d=_[424];var h=_[425];var p=_[426];var f=_[427];var g=_[428];var m=_[429];var v=_[430];var y=_[431];var w=_[432];var k=_[433];var x=_[434];var z=_[435];var b=_[436];var j=_[437];var a=_[438];var q=_[439];var r=_[440];var s=_[441];var i=_[442];var o=_[443];var u=_[444];var l=_[445];var c=_[446];var d=_[447];var h=_[448];var p=_[449];var f=_[450];var g=_[451];var m=_[452];var v=_[453];var y=_[454];var w=_[455];var k=_[456];var x=_[457];var z=_[458];var b=_[459];var j=_[460];var a=_[461];var q=_[462];var r=_[463];var s=_[464];var i=_[465];var o=_[466];var u=_[467];var l=_[468];var c=_[469];var d=_[470];var h=_[471];var p=_[472];var f=_[473];var g=_[474];var m=_[475];var v=_[476];var y=_[477];var w=_[478];var k=_[479];var x=_[480];var z=_[481];var b=_[482];var j=_[483];var a=_[484];var q=_[485];var r=_[486];var s=_[487];var i=_[488];var o=_[489];var u=_[490];var l=_[491];var c=_[492];var d=_[493];var h=_[494];var p=_[495];var f=_[496];var g=_[497];var m=_[498];var v=_[499];var y=_[500];var w=_[501];var k=_[502];var x=_[503];var z=_[504];var b=_[505];var j=_[506];var a=_[507];var q=_[508];var r=_[509];var s=_[510];var i=_[511];var o=_[512];var u=_[513];var l=_[514];var c=_[515];var d=_[516];var h=_[517];var p=_[518];var f=_[519];var g=_[520];var m=_[521];var v=_[522];var y=_[523];var w=_[524];var k=_[525];var x=_[526];var z=_[527];var b=_[528];var j=_[529];var a=_[530];var q=_[531];var r=_[532];var s=_[533];var i=_[534];var o=_[535];var u=_[536];var l=_[537];var c=_[538];var d=_[539];var h=_[540];var p=_[541];var f=_[542];var g=_[543];var m=_[544];var v=_[545];var y=_[546];var w=_[547];var k=_[548];var x=_[549];var z=_[550];var b=_[551];var j=_[552];var a=_[553];var q=_[554];var r=_[555];var s=_[556];var i=_[557];var o=_[558];var u=_[559];var l=_[560];var c=_[561];var d=_[562];var h=_[563];var p=_[564];var f=_[565];var g=_[566];var m=_[567];var v=_[568];var y=_[569];var w=_[570];var k=_[571];var x=_[572];var z=_[573];var b=_[574];var j=_[575];var a=_[576];var q=_[577];var r=_[578];var s=_[579];var i=_[580];var o=_[581];var u=_[582];var l=_[583];var c=_[584];var d=_[585];var h=_[586];var p=_[587];var f=_[588];var g=_[589];var m=_[590];var v=_[591];var y=_[592];var w=_[593];var k=_[594];var x=_[595];var z=_[596];var b=_[597];var j=_[598];var a=_[599];var q=_[600];var r=_[601];var s=_[602];var i=_[603];var o=_[604];var u=_[605];var l=_[606];var c=_[607];var d=_[608];var h=_[609];var p=_[610];var f=_[611];var g=_[612];var m=_[613];var v=_[614];var y=_[615];var w=_[616];var k=_[617];var x=_[618];var z=_[619];var b=_[620];var j=_[621];var a=_[622];var q=_[623];var r=_[624];var s=_[625];var i=_[626];var o=_[627];var u=_[628];var l=_[629];var c=_[630];var d=_[631];var h=_[632];var p=_[633];var f=_[634];var g=_[635];var m=_[636];var v=_[637];var y=_[638];var w=_[639];var k=_[640];var x=_[641];var z=_[642];var b=_[643];var j=_[644];var a=_[645];var q=_[646];var r=_[647];var s=_[648];var i=_[649];var o=_[650];var u=_[651];var l=_[652];var c=_[653];var d=_[654];var h=_[655];var p=_[656];var f=_[657];var g=_[658];var m=_[659];var v=_[660];var y=_[661];var w=_[662];var k=_[663];var x=_[664];var z=_[665];var b=_[666];var j=_[667];var a=_[668];var q=_[669];var r=_[670];var s=_[671];var i=_[672];var o=_[673];var u=_[674];var l=_[675];var c=_[676];var d=_[677];var h=_[678];var p=_[679];var f=_[680];var g=_[681];var m=_[682];var v=_[683];var y=_[684];var w=_[685];var k=_[686];var x=_[687];var z=_[688];var b=_[689];var j=_[690];var a=_[691];var q=_[692];var r=_[693];var s=_[694];var i=_[695];var o=_[696];var u=_[697];var l=_[698];var c=_[699];var d=_[700];var h=_[701];var p=_[702];var f=_[703];var g=_[704];var m=_[705];var v=_[706];var y=_[707];var w=_[708];var k=_[709];var x=_[710];var z=_[711];var b=_[712];var j=_[713];var a=_[714];var q=_[715];var r=_[716];var s=_[717];var i=_[718];var o=_[719];var u=_[720];var l=_[721];var c=_[722];var d=_[723];var h=_[724];var p=_[725];var f=_[726];var g=_[727];var m=_[728];var v=_[729];var y=_[730];var w=_[731];var k=_[732];var x=_[733];var z=_[734];var b=_[735];var j=_[736];var a=_[737];var q=_[738];var r=_[739];var s=_[740];var i=_[741];var o=_[742];var u=_[743];var l=_[744];var c=_[745];var d=_[746];var h=_[747];var p=_[748];var f=_[749];var g=_[750];var m=_[751];var v=_[752];var y=_[753];var w=_[754];var k=_[755];var x=_[756];var z=_[757];var b=_[758];var j=_[759];var a=_[760];var q=_[761];var r=_[762];var s=_[763];var i=_[764];var o=_[765];var u=_[766];var l=_[767];var c=_[768];var d=_[769];var h=_[770];var p=_[771];var f=_[772];var g=_[773];var m=_[774];var v=_[775];var y=_[776];var w=_[777];var k=_[778];var x=_[779];var z=_[780];var b=_[781];var j=_[782];var a=_[783];var q=_[784];var r=_[785];var s=_[786];var i=_[787];var o=_[788];var u=_[789];var l=_[790];var c=_[791];var d=_[792];var h=_[793];var p=_[794];var f=_[795];var g=_[796];var m=_[797];var v=_[798];var y=_[799];var w=_[800];var k=_[801];var x=_[802];var z=_[803];var b=_[804];var j=_[805];var a=_[806];var q=_[807];var r=_[808];var s=_[809];var i=_[8010];var o=_[8011];var u=_[8012];var l=_[8013];var c=_[8014];var d=_[8015];var h=_[8016];var p=_[8017];var f=_[8018];var g=_[8019];var m=_[8020];var v=_[8021];var y=_[8022];var w=_[8023];var k=_[8024];var x=_[8025];var z=_[8026];var b=_[8027];var j=_[8028];var a=_[8029];var q=_[8030];var r=_[8031];var s=_[8032];var i=_[8033];var o=_[8034];var u=_[8035];var l=_[8036];var c=_[8037];var d=_[8038];var h=_[8039];var p=_[8040];var f=_[8041];var g=_[8042];var m=_[8043];var v=_[8044];var y=_[8045];var w=_[8046];var k=_[8047];var x=_[8048];var z=_[8049];var b=_[8050];var j=_[8051];var a=_[8052];var q=_[8053];var r=_[8054];var s=_[8055];var i=_[8056];var o=_[8057];var u=_[8058];var l=_[8059];var c=_[8060];var d=_[8061];var h=_[8062];var p=_[8063];var f=_[8064];var g=_[8065];var m=_[8066];var v=_[8067];var y=_[8068];var w=_[8069];var k=_[8070];var x=_[8071];var z=_[8072];var b=_[8073];var j=_[8074];var a=_[8075];var q=_[8076];var r=_[8077];var s=_[8078];var i=_[8079];var o=_[8080];var u=_[8081];var l=_[8082];var c=_[8083];var d=_[8084];var h=_[8085];var p=_[8086];var f=_[8087];var g=_[8088];var m=_[8089];var v=_[8090];var y=_[8091];var w=_[8092];var k=_[8093];var x=_[8094];var z=_[8095];var b=_[8096];var j=_[8097];var a=_[8098];var q=_[8099];var r=_[8100];var s=_[8101];var i=_[8102];var o=_[8103];var u=_[8104];var l=_[8105];var c=_[8106];var d=_[8107];var h=_[8108];var p=_[8109];var f=_[8110];var g=_[8111];var m=_[8112];var v=_[8113];var y=_[8114];var w=_[8115];var k=_[8116];var x=_[8117];var z=_[8118];var b=_[8119];var j=_[8120];var a=_[8121];var q=_[8122];var r=_[8123];var s=_[8124];var i=_[8125];var o=_[8126];var u=_[8127];var l=_[8128];var c=_[8129];var d=_[8130];var h=_[8131];var p=_[8132];var f=_[8133];var g=_[8134];var m=_[8135];var v=_[8136];var y=_[8137];var w=_[8138];var k=_[8139];var x=_[8140];var z=_[8141];var b=_[8142];var j=_[8143];var a=_[8144];var q=_[8145];var r=_[8146];var s=_[8147];var i=_[8148];var o=_[8149];var u=_[8150];var l=_[8151];var c=_[8152];var d=_[8153];var h=_[8154];var p=_[8155];var f=_[8156];var g=_[8157];var m=_[8158];var v=_[8159];var y=_[8160];var w=_[8161];var k=_[8162];var x=_[8163];var z=_[8164];var b=_[8165];var j=_[8166];var a=_[8167];var q=_[8168];var r=_[8169];var s=_[8170];var i=_[8171];var o=_[8172];var u=_[8173];var l=_[8174];var c=_[8175];var d=_[8176];var h=_[8177];var p=_[8178];var f=_[8179];var g=_[8180];var m=_[8181];var v=_[8182];var y=_[8183];var w=_[8184];var k=_[8185];var x=_[8186];var z=_[8187];var b=_[8188];var j=_[8189];var a=_[8190];var q=_[8191];var r=_[8192];var s=_[8193];var i=_[8194];var o=_[8195];var u=_[8196];var l=_[8197];var c=_[8198];var d=_[8199];var h=_[8200];var p=_[8201];var f=_[8202];var g=_[8203];var m=_[8204];var v=_[8205];var y=_[8206];var w=_[8207];var k=_[8208];var x=_[8209];var z=_[8210];var b=_[8211];var j=_[8212];var a=_[8213];var q=_[8214];var r=_[8215];var s=_[8216];var i=_[8217];var o=_[8218];var u=_[8219];var l=_[8220];var c=_[8221];var d=_[8222];var h=_[8223];var p=_[8224];var f=_[8225];var g=_[8226];var m=_[8227];var v=_[8228];var y=_[8229];var w=_[8230];var k=_[8231];var x=_[8232];var z=_[8233];var b=_[8234];var j=_[8235];var a=_[8236];var q=_[8237];var r=_[8238];var s=_[8239];var i=_[8240];var o=_[8241];var u=_[8242];var l=_[8243];var c=_[8244];var d=_[8245];var h=_[8246];var p=_[8247];var f=_[8248];var g=_[8249];var m=_[8250];var v=_[8251];var y=_[8252];var w=_[8253];var k=_[8254];var x=_[8255];var z=_[8256];var b=_[8257];var j=_[8258];var a=_[8259];var q=_[8260];var r=_[8261];var s=_[8262];var i=_[8263];var o=_[8264];var u=_[8265];var l=_[8266];var c=_[8267];var d=_[8268];var h=_[8269];var p=_[8270];var f=_[8271];var g=_[8272];var m=_[8273];var v=_[8274];var y=_[8275];var w=_[8276];var k=_[8277];var x=_[8278];var z=_[8279];var b=_[8280];var j=_[8281];var a=_[8282];var q=_[8283];var r=_[8284];var s=_[8285];var i=_[8286];var o=_[8287];var u=_[8288];var l=_[8289];var c=_[8290];var d=_[8291];var h=_[8292];var p=_[8293];var f=_[8294];var g=_[8295];var m=_[8296];var v=_[8297];var y=_[8298];var w=_[8299];var k=_[8300];var x=_[8301];var z=_[8302];var b=_[8303];var j=_[8304];var a=_[8305];var q=_[8306];var r=_[8307];var s=_[8308];var i=_[8309];var o=_[8310];var u=_[8311];var l=_[8312];var c=_[8313];var d=_[8314];var h=_[8315];var p=_[8316];var f=_[8317];var g=_[8318];var m=_[8319];var v=_[8320];var y=_[8321];var w=_[8322];var k=_[8323];var x=_[8324];var z=_[8325];var b=_[8326];var j=_[8327];var a=_[8328];var q=_[8329];var r=_[8330];var s=_[8331];var i=_[8332];var o=_[8333];var u=_[8334];var l=_[8335];var c=_[8336];var d=_[8337];var h=_[8338];var p=_[8339];var f=_[8340];var g=_[8341];var m=_[8342];var v=_[8343];var y=_[8344];var w=_[8345];var k=_[8346];var x=_[8347];var z=_[8348];var b=_[8349];var j=_[8350];var a=_[8351];var q=_[8352];var r=_[8353];var s=_[8354];var i=_[8355];var o=_[8356];var u=_[8357];var l=_[8358];var c=_[8359];var d=_[8360];var h=_[8361];var p=_[8362];var f=_[8363];var g=_[8364];var m=_[8365];var v=_[8366];var y=_[8367];var w=_[8368];var k=_[8369];var x=_[8370];var z=_[8371];var b=_[8372];var j=_[8373];var a=_[8374];var q=_[8375];var r=_[8376];var s=_[8377];var i=_[8378];var o=_[8379];var u=_[8380];var l=_[8381];var c=_[8382];var d=_[8383];var h=_[8384];var p=_[8385];var f=_[8386];var g=_[8387];var m=_[8388];var v=_[8389];var y=_[8390];var w=_[8391];var k=_[8392];var x=_[8393];var z=_[8394];var b=_[8395];var j=_[8396];var a=_[8397];var q=_[8398];var r=_[8399];var s=_[8400];var i=_[8401];var o=_[8402];var u=_[8403];var l=_[8404];var c=_[8405];var d=_[8406];var h=_[8407];var p=_[8408];var f=_[8409];var g=_[8410];var m=_[8411];var v=_[8412];var y=_[8413];var w=_[8414];var k=_[8415];var x=_[8416];var z=_[8417];var b=_[8418];var j=_[8419];var a=_[8420];var q=_[8421];var r=_[8422];var s=_[8423];var i=_[8424];var o=_[8425];var u=_[8426];var l=_[8427];var c=_[8428];var d=_[8429];var h=_[8430];var p=_[8431];var f=_[8432];var g=_[8433];var m=_[8434];var v=_[843
```

Esquema Visual

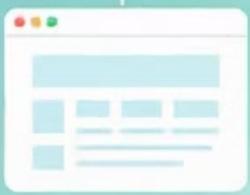
El siguiente diagrama muestra de forma conceptual cómo funciona el paso de parámetros en Java y qué ocurre en memoria en cada caso.



ⓘ Interpretación del esquema:

- En el caso de **primitivos**, la flecha representa una copia de datos, completamente independiente.
- En el caso de **objetos**, se copia la dirección de memoria (referencia).
- Si se accede al objeto y se cambian sus atributos, ambos apuntan al mismo objeto y el cambio se propaga.
- Si dentro del método reasignamos la variable, esa nueva referencia solo afecta al ámbito local del método.

Este modelo es lo que denominamos "**paso de referencias por valor**", una expresión más precisa que evita confusiones con el verdadero "paso por referencia" que existe en otros lenguajes como C++ o C#.



Caso de Estudio: Google Docs y las referencias compartidas

Contexto

Google Docs, la aplicación de edición colaborativa de Google, permite que varias personas editen un mismo documento simultáneamente. Todos los usuarios ven los cambios en tiempo real sin duplicar el archivo completo en cada dispositivo.

Estrategia

Para lograr esto, el sistema usa **referencias compartidas** a un único objeto de documento almacenado en la nube. Cada usuario tiene una "vista local" que se sincroniza con el objeto principal. Técnicamente, el cliente no posee una copia completa del documento; tiene una **referencia sincronizada** que apunta al documento en el servidor.

Cada vez que un usuario escribe o borra texto, su cliente envía una modificación sobre **la referencia del objeto compartido**, y el servidor actualiza el estado global, propagando los cambios al resto de usuarios.

Este comportamiento refleja exactamente lo que sucede en Java cuando un objeto se pasa como parámetro por valor de referencia: todos los métodos (o, en este caso, clientes) trabajan sobre el **mismo objeto**, modificando su estado interno, sin necesidad de crear copias completas.

Resultado

Gracias a este modelo de paso por referencia compartida, Google Docs logra:

- **Eficiencia:** no se duplican estructuras completas de datos.
- **Coherencia:** todos los usuarios ven el mismo estado actualizado.
- **Control:** el servidor mantiene la versión principal del objeto.

En el desarrollo de software, este mismo principio se aplica en Java cuando compartimos instancias entre métodos o módulos. Es potente, pero exige cuidado para evitar efectos indeseados cuando varios métodos modifican el mismo objeto sin coordinación.

Herramientas y Consejos

Consejos prácticos

1 Recuerda que en Java todo se pasa por valor

Lo que cambia es *qué* se copia: el valor o la referencia. No confundas esto con el paso por referencia real, que Java no soporta.

Crea copias cuando quieras proteger el estado. Si no deseas modificar el objeto original dentro de un método, usa un clon o un nuevo objeto.

```
Persona copia = new  
Persona(original.nombre);
```

2 Evita efectos secundarios

Un método debe tener un propósito claro. Si modifica el estado de objetos globales sin advertirlo, puede causar comportamientos impredecibles.

3 Utiliza objetos inmutables

Clases como String, LocalDate, BigDecimal son inmutables: cualquier "modificación" devuelve un nuevo objeto. Esto elimina los errores derivados del paso de referencias compartidas.

4 Devuelve resultados en lugar de modificar parámetros

Es preferible retornar un nuevo objeto o valor, lo que facilita la comprensión del código.

Herramientas recomendadas

Java Visualizer

(pythontutor.com/java.html): permite ver gráficamente cómo se almacenan las variables y objetos en memoria, ideal para comprender referencias.

Debugger de IntelliJ IDEA

puedes inspeccionar el contenido de cada variable y ver qué referencias apuntan al mismo objeto.

Replit (modo "memory trace")

visualiza paso a paso el comportamiento del código en ejecución.

Eclipse Memory Analyzer

útil para proyectos grandes; permite detectar fugas de memoria y entender la distribución de referencias.

Estas herramientas son muy valiosas en entornos de aprendizaje y en equipos de desarrollo profesional donde la depuración es parte diaria del trabajo.

Mitos y Realidades

 **Mito:** "Java pasa objetos por referencia."

→ **FALSO.** Java pasa **copias de referencias**, no referencias directas. Esto significa que el método recibe una copia del puntero (dirección de memoria). Si cambias la referencia dentro del método, el original no se ve afectado.

 **Realidad:** El paso por valor se mantiene siempre, tanto para primitivos como para objetos. Lo que cambia es el contenido copiado (el valor o la referencia).

 **Mito:** "Los Strings se comportan igual que los demás objetos."

→ **FALSO.** Aunque String es una clase (por tanto, un objeto), es **inmutable**. Si "modificas" un String, en realidad estás creando uno nuevo, no alterando el existente. Por ejemplo:

```
String s = "Hola";
modificar(s); // dentro se hace s = s + "mundo";
```

El objeto original "Hola" sigue existiendo; el método ha creado un nuevo "Hola mundo", pero no ha afectado al original.

 **Realidad:** La inmutabilidad evita los efectos del paso por referencia simulada, garantizando seguridad y consistencia en el manejo de textos.

Resumen Final

- **Java pasa todo por valor.**
- **Primitivos:** se copia el valor; no afecta al original.
- **Objetos:** se copia la referencia; modificar propiedades afecta, reasignar no.
- **Evita efectos colaterales:** usa objetos inmutables o devuelve nuevos valores.
- **Comprender la diferencia** entre copiar valor y copiar referencia es esencial para escribir código confiable y mantener la coherencia del programa.

Sesión 15: Sobrecarga de métodos y constructores

La **sobrecarga (overloading)** es un principio esencial de la **programación orientada a objetos (POO)** que permite reutilizar nombres de métodos o constructores con **diferentes firmas** (conjuntos de parámetros). En otras palabras, Java permite tener **varios métodos con el mismo nombre** siempre que **difieran en el número o tipo de argumentos**.

Esta característica mejora la legibilidad y flexibilidad del código, ya que el mismo método puede adaptarse a distintas situaciones sin obligar a crear nombres redundantes o poco claros.

¿Qué significa "diferente firma"?

La **firma** de un método está compuesta por su **nombre y los tipos y orden de los parámetros**. El **tipo de retorno no forma parte de la firma**, por lo que **no puede haber dos métodos con la misma firma y distinto tipo de retorno**.

Ejemplo básico:

```
int sumar(int a, int b) { return a + b; }
double sumar(double a, double b) { return a + b; }
```

Aquí tenemos dos métodos `sumar()` con el mismo nombre, pero uno recibe enteros y el otro números decimales (`double`). Cuando llamas a `sumar(2, 3)`, el compilador elige automáticamente la versión más adecuada según los tipos de los argumentos.

Este proceso se conoce como **resolución de sobrecarga (overload resolution)** y ocurre **en tiempo de compilación**, no en tiempo de ejecución. Por tanto, no se trata de polimorfismo dinámico (como la **sobrescritura**, `@Override`), sino de **polimorfismo estático**.

Sobrecarga de constructores

El concepto también se aplica a los **constructores**, lo que permite crear objetos de una clase con diferentes inicializaciones.

Ejemplo:

```
class Rectangulo {  
    int ancho, alto;  
  
    // Constructor 1: cuadrado  
    Rectangulo(int lado) {  
        ancho = alto = lado;  
    }  
    // Constructor 2: rectángulo  
    Rectangulo(int a, int b) {  
        ancho = a;  
        alto = b;  
    }  
}
```

Este diseño ofrece flexibilidad: el programador puede crear un Rectangulo cuadrado o rectangular según los parámetros que use.

La **sobrecarga** se utiliza en prácticamente todas las librerías estándar de Java. Por ejemplo, la clase PrintStream (usada por System.out) tiene múltiples versiones del método println() que aceptan int, double, String, char, etc.

Ventajas de la sobrecarga

- **Reutilización de código:** no necesitas crear nombres distintos para variaciones del mismo método.
- **Legibilidad:** los métodos mantienen una intención clara y coherente.
- **Flexibilidad:** un mismo método puede adaptarse a distintos tipos de entrada.
- **Mantenibilidad:** si el comportamiento general cambia, solo se actualiza la lógica común.

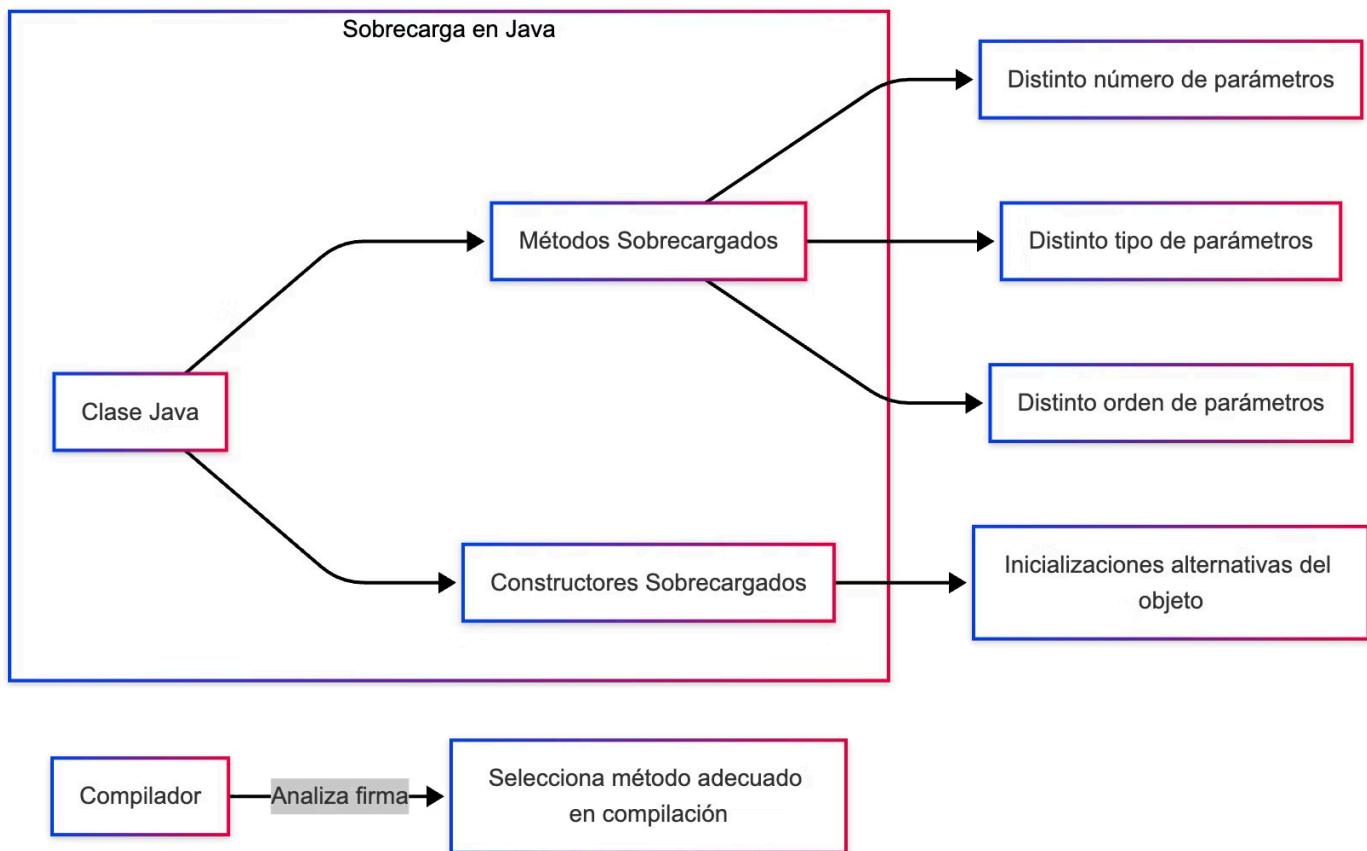
Peligros y limitaciones

- Una sobrecarga mal diseñada puede generar **confusión** o **ambigüedad** si las firmas son demasiado parecidas.
- Si las diferencias de parámetros son mínimas (por ejemplo, entre int y Integer), el compilador podría generar advertencias o comportamientos inesperados debido al **autoboxing**.

Por tanto, la sobrecarga debe usarse con criterio y claridad semántica.

Esquema Visual

A continuación se presenta un esquema conceptual de cómo se organiza la sobrecarga en Java:



Interpretación del esquema:

- Diferencias válidas
 - En una clase pueden coexistir varios métodos con el mismo nombre, siempre que difieran en **número, tipo o orden** de parámetros.
- Constructores múltiples
 - Lo mismo aplica a **constructores**, permitiendo distintas formas de inicializar un objeto.
- Resolución automática
 - Durante la **compilación**, el compilador selecciona automáticamente el método correcto según la firma más específica.

Esto hace que el proceso sea eficiente y predecible, ya que no se requiere resolver en tiempo de ejecución.

Caso de Estudio: Instagram y la Sobrecarga en sus Clases de Medios

Contexto

Instagram gestiona millones de archivos multimedia (imágenes, vídeos, streams en directo) que los usuarios suben cada día. Cada uno de estos tipos requiere un tratamiento diferente, pero el objetivo general —“crear un nuevo medio”— es el mismo.

Estrategia

En su API interna, Instagram implementa una clase de tipo Media que utiliza **sobrecarga de constructores y métodos** para simplificar el proceso de creación y publicación de contenido.

Ejemplo conceptual (simplificado):

```
class Media {  
    Media(String rutalmagen) { /* carga imagen local */ }  
    Media(File archivoVideo) { /* carga video */ }  
    Media(InputStream stream) { /* carga contenido en directo */ }  
}
```

De esta manera, el desarrollador puede crear un nuevo objeto Media con la misma llamada, pero pasando parámetros diferentes según el tipo de contenido.

También pueden existir métodos sobrecargados para procesar los medios:

```
public void publicar(String mensaje)  
{ /* post con texto */ }  
public void publicar(String mensaje,  
String hashtag) { /* post con texto + hashtag */ }  
public void publicar(String mensaje,  
Location ubicacion) { /* post geolocalizado */ }
```

Resultado

Gracias a la sobrecarga, Instagram logra:

- **Simplificar el uso de su API:** el desarrollador no necesita memorizar múltiples nombres de métodos.
- **Adaptar la funcionalidad:** distintos formatos y tipos de contenido se procesan con el mismo flujo de trabajo.
- **Mejorar la legibilidad:** el código es coherente y autoexplicativo.

Este caso ilustra cómo el principio de sobrecarga, bien aplicado, no solo mejora la experiencia del programador, sino también la eficiencia de las librerías que utilizan millones de usuarios.

Herramientas y Consejos

Consejos prácticos

1 Cambia el número o tipo de parámetros, no el tipo de retorno

Dos métodos con igual nombre y mismos parámetros pero distinto tipo de retorno provocan error de compilación.

```
// INCORRECTO
int calcular() { return 1; }
double calcular() { return 1.0; } //
Error: misma firma
```

2 Evita la sobrecarga excesiva
Más de cuatro versiones del mismo método tienden a confundir. Si necesitas muchas variaciones, considera usar **parámetros opcionales** con valores por defecto (en Java, simulables con overloading + null o Optional).

3 Usa this() dentro de constructores

Permite encadenar constructores y evitar duplicación de código.

```
class Libro {
    String titulo; int paginas;
    Libro(String titulo) {
        this(titulo, 100); // llama al otro
        constructor
    }
    Libro(String titulo, int paginas) {
        this.titulo = titulo; this.paginas =
        paginas;
    }
}
```

4 Documenta claramente las versiones

Utiliza `/** comentarios Javadoc */` para especificar qué hace cada sobrecarga.

```
/**
 * Calcula el área de un rectángulo.
 * @param ancho ancho en metros
 * @param alto alto en metros
 */
double area(double ancho, double
alto) {...}
```

5 Prueba las versiones con unit testing

Frameworks como JUnit o TestNG te permiten verificar que cada sobrecarga funciona como se espera con distintos tipos de datos.

Herramientas útiles

IntelliJ IDEA / Eclipse "Refactor → Overload Method"

crea automáticamente versiones sobrecargadas del mismo método.

Javadoc Generator

genera documentación estructurada de las diferencias entre métodos.

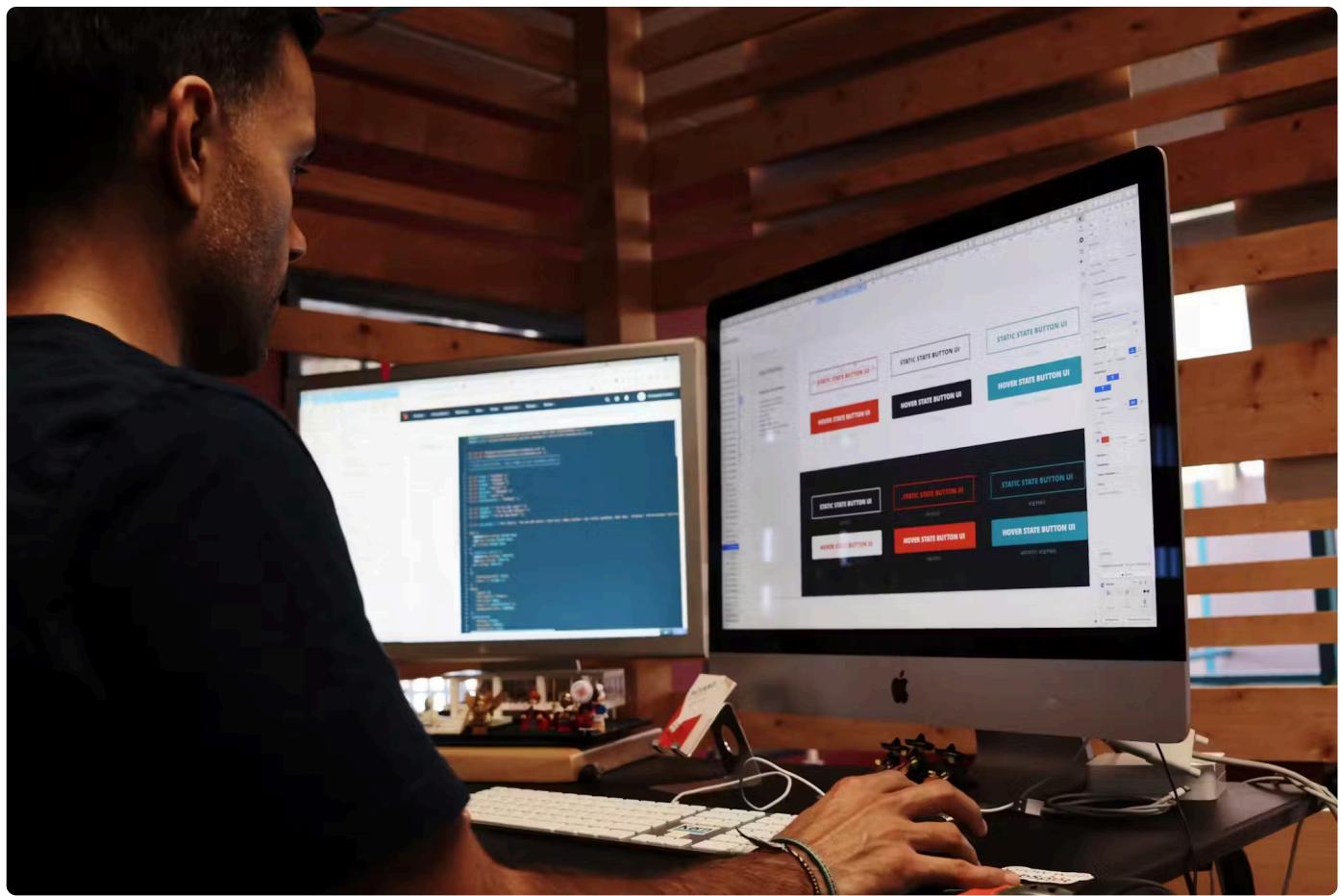
SonarLint o PMD

detectan duplicidades o sobrecargas ambiguas que pueden causar confusión.

JDoodle o Replit

ideales para probar ejemplos rápidos de sobrecarga.

Estas herramientas son habituales en entornos de desarrollo profesional y ayudan a mantener la consistencia del código y su claridad.



Mitos y Realidades

 **Mito:** "La sobrecarga depende del tipo de retorno."

→ **FALSO.** El tipo de retorno **no forma parte de la firma** de un método. Dos métodos con los mismos parámetros pero diferente tipo de retorno causan error de compilación.

 **Realidad:** Solo los **parámetros** (tipo, número, orden) determinan la sobrecarga válida.

 **Mito:** "La sobrecarga se decide en tiempo de ejecución."

→ **FALSO.** Esa afirmación corresponde a la **sobrescritura (overriding)**. En la sobrecarga, el método correcto se elige **en tiempo de compilación** según la coincidencia más específica de parámetros.

 **Realidad:** La sobrecarga es **polimorfismo estático**, mientras que la sobrescritura es **polimorfismo dinámico**.

Resumen Final

- **Sobrecarga = mismo nombre, distintos parámetros.**
- **Aplica a métodos y constructores.**
- **Se resuelve en tiempo de compilación (polimorfismo estático).**
- **No depende del tipo de retorno.**
- **Usa this() para encadenar constructores y evitar duplicación.**
- Documenta y limita el número de sobrecargas para mantener claridad.



Sesión 16: Modularización de programas grandes

Cuando los programas crecen en tamaño y complejidad, mantenerlos organizados se convierte en un desafío real. El código monolítico —aquel en el que todo se encuentra en un único bloque o archivo gigante— termina siendo inmanejable: es difícil de mantener, de probar, y los cambios en una parte pueden afectar a otras de manera impredecible. Para evitar este caos surge la **modularización**, un principio fundamental de la ingeniería de software que consiste en dividir un programa en **módulos coherentes y autónomos**, cada uno responsable de una función específica.

En el caso de **Java**, este enfoque se implementa mediante dos mecanismos complementarios:

Paquetes (packages)

desde las primeras versiones de Java, los paquetes permiten agrupar clases relacionadas bajo un mismo espacio de nombres, evitando conflictos y promoviendo la organización lógica. Por ejemplo:

```
com.miempresa.usuarios
com.miempresa_pedidos
com.miempresa.utilidades
```

Cada paquete contiene clases que comparten una finalidad común, como la gestión de usuarios o el procesamiento de pedidos.

Sistema de módulos (Java Platform Module System – JPMS)

introducido en **Java 9**, da un paso más allá al permitir definir explícitamente las dependencias entre módulos y qué partes del código son visibles desde fuera. Esto se logra mediante el archivo `module-info.java`, donde se especifica qué paquetes se exportan y qué otros módulos se requieren. Ejemplo:

```
module com.miempresa.usuarios {
    exports
        com.miempresa.usuarios.modelo;
    requires com.miempresa.utilidades;
}
```

Con esta estructura, el código deja de ser un conjunto de clases dispersas y se convierte en un sistema ordenado y escalable.

El propósito de modularizar no es solo **ordenar** el código, sino **hacerlo más robusto y reutilizable**. En proyectos grandes, la modularización:

Mejora la mantenibilidad, ya que los cambios en un módulo no afectan a los demás si la interfaz pública está bien definida.

Aumenta la reutilización, permitiendo usar módulos en distintos proyectos sin duplicar código.

Facilita el trabajo en equipo, al permitir que diferentes desarrolladores trabajen en módulos independientes.

Refuerza la seguridad y la encapsulación, al exponer únicamente lo necesario.

Estos beneficios descansan sobre dos principios universales:

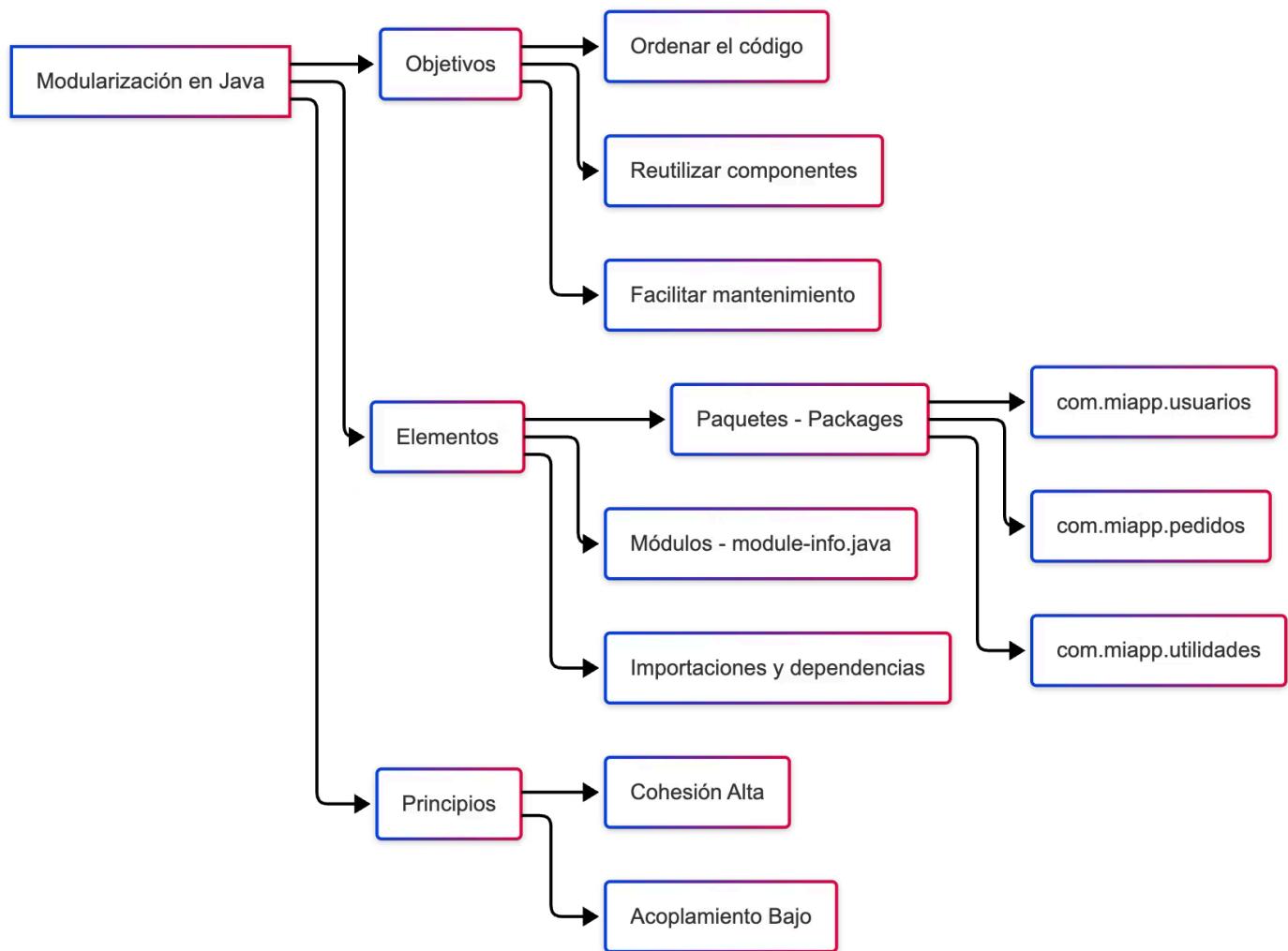
Alta cohesión: cada módulo debe encargarse de una única responsabilidad o conjunto estrechamente relacionado de funciones.

Bajo acoplamiento: los módulos deben depender lo menos posible entre sí; cuando se comunican, deben hacerlo mediante interfaces bien definidas.

En definitiva, modularizar no solo es una cuestión de estilo, sino una estrategia esencial para construir **software sostenible, escalable y fácil de evolucionar**.



Esquema visual:



ⓘ Descripción detallada:

- El **nodo central** “Modularización en Java” es el objetivo global de organizar el software por funcionalidades.
- La rama de **Objetivos** destaca: ordenar el código, reutilizar componentes y facilitar el mantenimiento.
- La rama de **Elementos** detalla las estructuras clave:
 - **Paquetes (packages)**: Unidades básicas de organización.
 - **Módulos (module-info.java)**: Definen qué se exporta y qué se oculta.
 - **Importaciones**: Relaciones explícitas entre módulos.
- La rama de **Principios** subraya un buen diseño modular: alta cohesión (módulo con una responsabilidad) y bajo acoplamiento (mínima dependencia).
- Los subnodos (ej. com.miapp.usuarios) ejemplifican la organización típica de un proyecto profesional.

Este esquema refleja la lógica jerárquica del diseño modular: **de lo general a lo específico**, de la arquitectura al detalle de implementación.

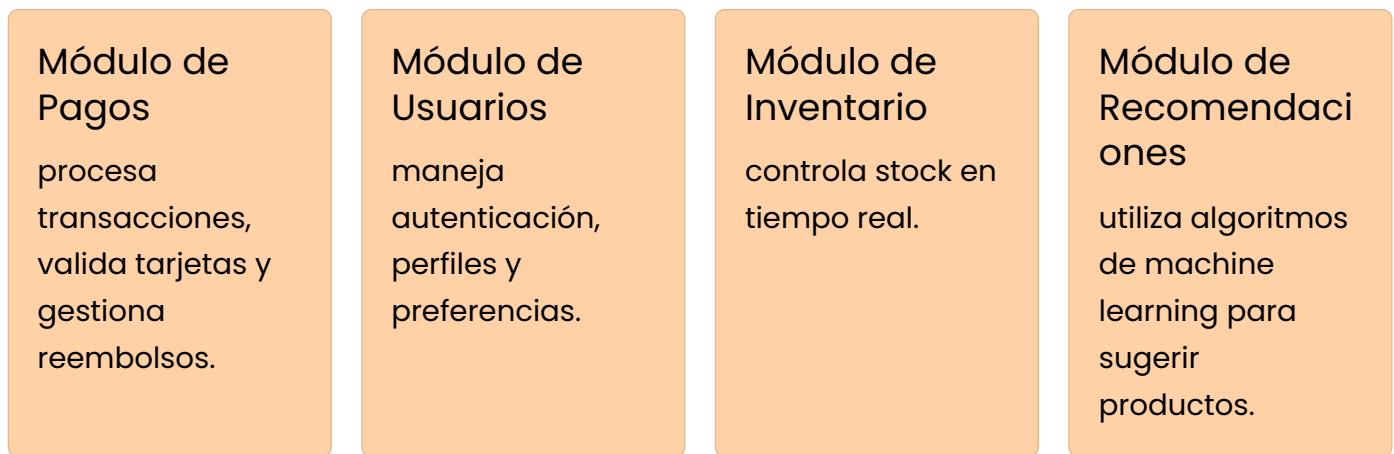
Caso de estudio: Amazon y la orquestación de miles de módulos

Contexto

Amazon gestiona uno de los ecosistemas de software más complejos del planeta. Su plataforma de comercio electrónico, sus sistemas logísticos y su infraestructura en la nube (AWS) dependen de millones de líneas de código distribuidas en miles de equipos y centros de datos.

Estrategia modular:

En lugar de mantener un sistema monolítico, Amazon adoptó una **arquitectura modular basada en microservicios**, donde cada módulo es responsable de una función específica y se comunica con otros mediante interfaces bien definidas (APIs). Algunos ejemplos:



Cada módulo está desarrollado, desplegado y escalado de forma independiente, siguiendo el principio de **cohesión alta y acoplamiento bajo**. Además, los equipos son pequeños y autónomos ("two-pizza teams"), lo que refuerza la modularidad organizativa y técnica.

Resultado

- Escalar sin perder estabilidad.
- Desplegar cambios cientos de veces al día sin interrumpir el servicio global.
- Reutilizar módulos en distintos contextos (por ejemplo, el sistema de pagos también se emplea en AWS Marketplace).
- Reducir drásticamente el tiempo de recuperación ante fallos.

Conclusión del caso: Amazon demuestra que la modularización es un **principio estratégico de negocio** clave, permitiendo una innovación constante y manteniendo la integridad del sistema.

Herramientas y consejos

Consejos prácticos:

1 Divide por funcionalidades, no por capas técnicas

No organices el proyecto en paquetes como "controllers", "models" o "utils" únicamente. Es preferible estructurarlo por dominios funcionales: usuarios, pedidos, pagos. Esto alinea la arquitectura técnica con la lógica del negocio.

2 Define interfaces claras

Cada módulo debe ofrecer una interfaz o conjunto de clases públicas que actúen como contrato con el exterior. Evita que otros módulos dependan de implementaciones internas.

3 Encapsula lo que no necesite ser público

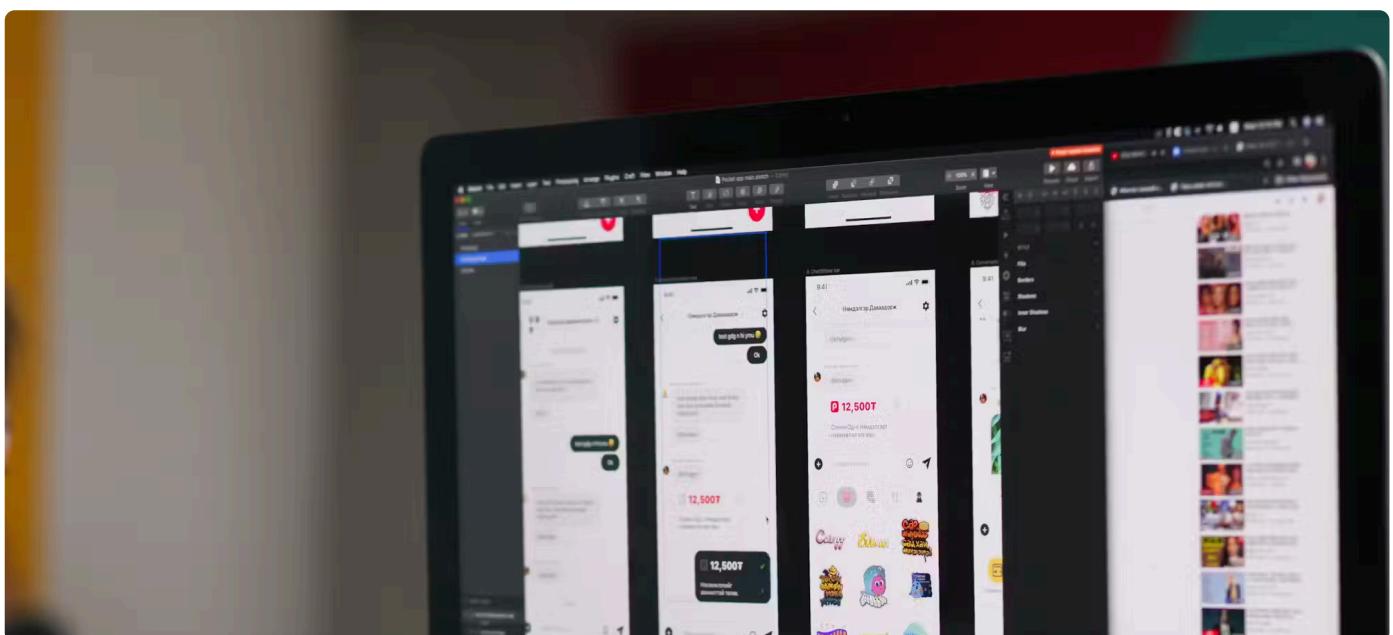
Utiliza modificadores de acceso (private, protected, package-private) para proteger las clases y métodos internos. Cuanta menos visibilidad, mayor independencia.

4 Documenta las dependencias

Mantén un archivo de documentación o un diagrama de dependencias actualizado. Esto previene errores de compilación y facilita la integración continua.

5 Aplica refactorización constante

A medida que el proyecto crece, los límites entre módulos pueden cambiar. Revisar y ajustar la estructura modular es parte natural del mantenimiento.



Herramientas útiles:

IntelliJ IDEA – Structure View

Permite visualizar jerárquicamente paquetes, clases y dependencias internas. Es ideal para reorganizar código de manera visual.

JDeps (Java Dependency Analyzer)

Herramienta incluida en el JDK que analiza dependencias entre paquetes y módulos. Detecta acoplamientos innecesarios y ayuda a preparar proyectos para el sistema modular de Java 9+.

Maven Modules / Gradle Multi-Project

Ambos sistemas de construcción soportan proyectos compuestos por varios módulos con dependencias explícitas. Permiten compilar, probar y desplegar cada módulo de forma independiente o conjunta.

ArchUnit

Biblioteca que permite definir y validar reglas arquitectónicas en código (por ejemplo, "el paquete util no puede depender de controller"). Ideal para mantener la integridad modular con tests automáticos.

Consejo profesional:

Antes de modularizar, crea un **mapa conceptual de responsabilidades** del sistema. Pregúntate: ¿qué funcionalidades podrían existir por separado sin romper la lógica general? Si la respuesta es afirmativa, probablemente tienes un candidato a módulo independiente.

Mitos y realidades sobre la modularización

 **Mito:** "Dividir el código en muchos paquetes hace el programa más lento."

 **Realidad:** La modularización afecta solo a la estructura lógica del código, no a su rendimiento en ejecución. El bytecode compilado se ejecuta igual de rápido; lo que mejora es la legibilidad y el mantenimiento.

 **Mito:** "Los módulos complican el trabajo en equipos pequeños."

 **Realidad:** Justamente lo contrario. La modularización distribuye responsabilidades y evita que varios desarrolladores trabajen sobre el mismo archivo. Incluso en equipos pequeños, permite desarrollar de forma más ordenada y escalable.

Resumen final

- **Modularización:** técnica que divide programas grandes en partes independientes y coherentes.
- **En Java:** se aplica mediante *packages* y *module-info.java*.
- **Principios clave:** cohesión alta (una sola responsabilidad) y acoplamiento bajo (mínima dependencia).
- **Ventajas:** mejor mantenimiento, reutilización, escalabilidad y trabajo colaborativo.
- **Herramientas útiles:** IntelliJ Structure View, JDeps, Maven Modules.
- **Ejemplo real:** Amazon demuestra cómo la modularidad impulsa la innovación y la estabilidad.



Sesión 17: Métodos estáticos y clases helper

En programación orientada a objetos, una de las decisiones más frecuentes que debes tomar es si un comportamiento debe pertenecer a **una instancia concreta de una clase** o a **la clase en sí misma**. Los **métodos estáticos** pertenecen al segundo caso: se asocian directamente a la clase, no a un objeto específico. Esto significa que **no necesitan ser instanciados** para ser utilizados, lo cual los hace perfectos para funciones generales, utilitarias o de cálculo.

En Java, los métodos estáticos se definen con la palabra clave static. Por ejemplo:

```
public class MathHelper {
    public static double areaCirculo(double r) {
        return Math.PI * r * r;
    }
}
```

Para usarlos, basta con llamar al método desde la clase, sin crear objetos:

```
double area = MathHelper.areaCirculo(5);
```

Aquí no existe un objeto de tipo MathHelper, sino que la función se ejecuta directamente desde la clase. Esto tiene una consecuencia importante: **todas las llamadas al método comparten una única copia en memoria**, lo que reduce la carga de instanciación y simplifica el acceso.

Sin embargo, esta potencia implica también una **limitación clave**: los métodos estáticos **no pueden acceder a variables de instancia ni a métodos no estáticos**. La razón es lógica: como no existe un objeto, no hay estado al que acceder. Por eso se dice que los métodos estáticos deben ser "**puros**", es decir, operaciones que siempre producen el mismo resultado para las mismas entradas y no dependen de ningún contexto interno.

Las **clases helper** (también conocidas como *utility classes*) surgen precisamente para agrupar este tipo de métodos. Se utilizan en prácticamente todos los proyectos Java para encapsular funciones comunes que no pertenecen a una entidad específica. Algunos ejemplos habituales:

Validaciones

`ValidationUtils.isEmailValid()`

Conversión de datos

`StringUtils.capitalize()`

Operaciones matemáticas

`Math.sqrt()`

Formateo de fechas

`DateUtils.formatDate()`

Su objetivo es **centralizar operaciones repetidas** y evitar duplicación de código. Al reunir funciones similares en una sola clase, se facilita la reutilización y el mantenimiento.

Ahora bien, abusar de ellas también puede ser un problema. Si todo el proyecto termina lleno de métodos estáticos, se pierde el principio de **encapsulación y extensibilidad** propio de la orientación a objetos. Por eso, una buena práctica profesional consiste en **equilibrar**: usar métodos estáticos para operaciones puras y helper classes para tareas utilitarias, pero mantener la lógica de negocio principal en clases instanciables que representen entidades reales.

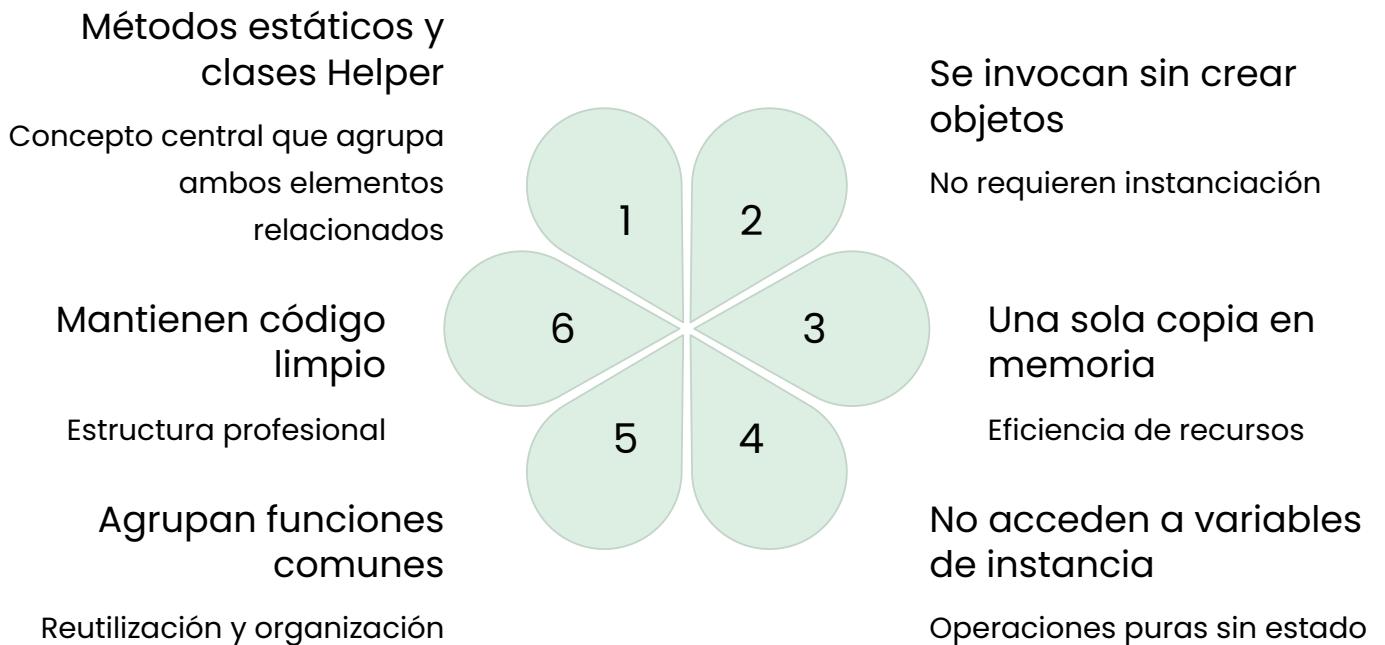
En resumen:

- **Usa métodos estáticos para operaciones sin estado.**
- **Agrúpalos en helpers coherentes por funcionalidad.**
- **Evita sobrecargar el proyecto con helpers genéricos que no reflejen el dominio del negocio.**

Esta distinción es lo que separa un código organizado y profesional de uno desordenado y difícil de mantener.

Esquema visual

A continuación se presenta un esquema conceptual de la relación entre **métodos estáticos**, **clases helper** y su función en el desarrollo profesional.



Descripción detallada del esquema:

Métodos estáticos

Características esenciales:

- **Sin instanciación:** no requieren crear objetos.
- **Una sola copia:** se almacenan una vez en memoria.
- **Sin estado:** no dependen de variables de instancia.
- **Puros:** devuelven siempre el mismo resultado con los mismos parámetros.

Clases helper

Ventajas destacadas:

- **Agrupan funciones comunes**
- **Fomentan la reutilización**
- **Organizan mejor el código**
- **Ubicación habitual:** paquete util o helpers

Este esquema refleja el papel de los helpers como **bibliotecas internas reutilizables** que dan soporte a la lógica principal del sistema sin formar parte directa del modelo de negocio.

Caso de estudio: Spotify y sus clases helper de utilidades internas

Contexto

Spotify, la plataforma de streaming musical más utilizada del mundo, maneja millones de operaciones por segundo: recomendaciones, cálculos de popularidad, generación de estadísticas de escucha, conversiones de formatos, etc. Para mantener un código modular y eficiente, Spotify organiza muchas de estas operaciones repetitivas en **clases helper con métodos estáticos**, dentro de su arquitectura basada en microservicios.

Estrategia técnica

En lugar de replicar código en cada microservicio, Spotify utiliza librerías compartidas llamadas **Utils** (de *utilities*) que incluyen cientos de funciones reutilizables. Algunos ejemplos de sus helpers:

MathUtils

contiene métodos de cálculo y normalización de valores, por ejemplo, para ponderar la relevancia de canciones en un algoritmo de recomendación.

TextUtils

gestiona limpieza de textos, eliminación de caracteres especiales y comparaciones insensibles a mayúsculas.

DateHelper

estandariza el formato de fechas entre servicios (UTC, ISO-8601).

NetworkHelper

comprueba el estado de conectividad o latencia entre servidores.

Cada helper agrupa **métodos estáticos puros**, garantizando que cualquier módulo de la empresa pueda utilizarlos sin dependencias complejas.

Resultado

- **Reutilización total:** cientos de servicios comparten la misma librería de utilidades.
- **Mantenimiento simplificado:** si se actualiza una función común (por ejemplo, un nuevo formato de fecha), el cambio se propaga automáticamente a todos los módulos.
- **Menos errores:** al centralizar funciones críticas, se evita la duplicación de lógica en distintos lugares.
- **Rendimiento optimizado:** al ser estáticos, los métodos se cargan una sola vez en memoria, reduciendo el coste de instanciación.

Conclusión del caso: Spotify demuestra cómo el uso correcto de helpers estáticos puede potenciar la **eficiencia, coherencia y escalabilidad del desarrollo en grandes equipos**. El secreto no está en usar muchos métodos estáticos, sino en agruparlos con sentido y responsabilidad.



Herramientas y consejos

Consejos profesionales:

- Usa métodos estáticos solo para operaciones puras

Si el resultado depende únicamente de los parámetros de entrada, sin alterar ni depender del estado del objeto, es un buen candidato a ser estático. Ejemplo: cálculos matemáticos, conversiones de formato, validaciones simples.

- Evita abusar de helpers

No todo debe ser estático. La orientación a objetos se basa en encapsular comportamiento dentro de instancias. Si un método depende del contexto o del estado (por ejemplo, el usuario actual o la sesión activa), debe ser parte de una clase instanciable.

- Agrupa helpers por dominio lógico

En lugar de tener una sola clase Utils gigantesca, crea varias helpers específicas: StringHelper, DateHelper, FileHelper. Esto mejora la cohesión y la legibilidad.

- Usa un paquete específico para las utilidades

Por convención, los proyectos profesionales agrupan estas clases en un paquete com.empres.util o helpers. Así se evita mezclar código de negocio con código de soporte.

- Acompaña siempre tus helpers con tests unitarios

Cada método estático debe estar cubierto por tests automáticos. Las pruebas unitarias garantizan que las funciones se comporten igual aunque cambie el entorno o la versión del código.

Herramientas:

- JUnit 5

Marco estándar de testing en Java. Permite probar fácilmente métodos estáticos con anotaciones como @Test y verificar su salida con aserciones (assertEquals, assertTrue, etc.).

- SonarLint

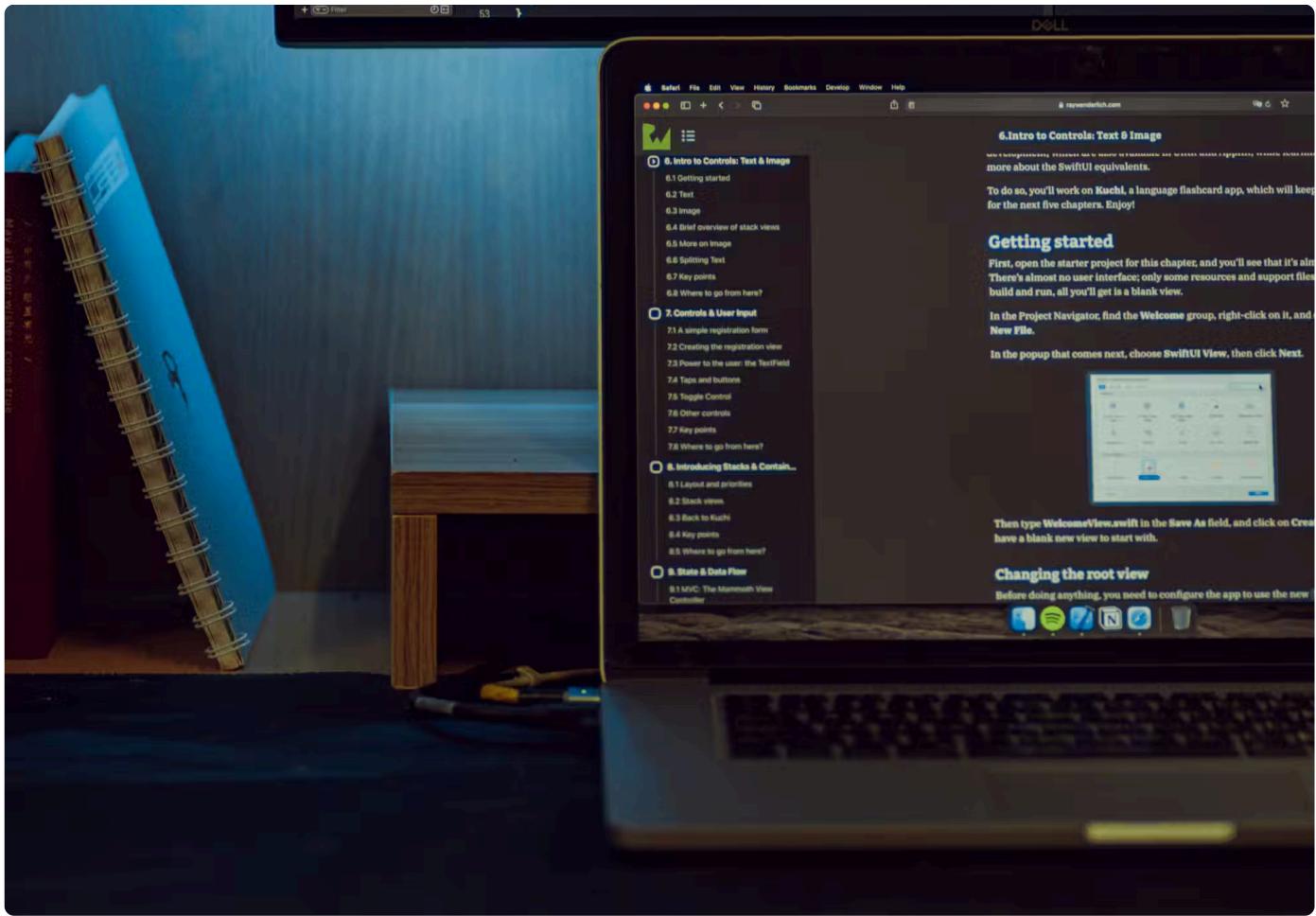
Plugin que analiza el código en tiempo real y alerta sobre posibles malas prácticas, como el exceso de métodos estáticos o la falta de encapsulación. Disponible para IntelliJ, VS Code y Eclipse.

- IntelliJ Code Analyzer

Herramienta integrada en IntelliJ IDEA que detecta dependencias estáticas excesivas, uso indebido de métodos static, y sugiere refactorizaciones automáticas.

- Checkstyle o PMD

Analizadores de código que permiten definir reglas para la documentación y el uso correcto de métodos estáticos.



❑ Ejemplo práctico de aplicación:

```
public final class ValidationHelper {  
    private ValidationHelper() {} // evita instanciación  
  
    public static boolean isValidEmail(String email) {  
        return email != null && email.matches("^[\\w.-]+@[\\w.-]+\\.\\w+$");  
    }  
  
    public static boolean isPositive(int n) {  
        return n > 0;  
    }  
}
```

El modificador final evita herencia y el constructor privado impide crear instancias, reforzando el propósito de la clase como *helper* pura. Ambos métodos son estáticos, puros y fácilmente testeables.

Mitos y Realidades

 **Mito:** "Los métodos estáticos son siempre mejores."

 **Realidad:** No. Son útiles cuando no hay necesidad de mantener estado o extender comportamiento. Pero si tu método podría cambiar según el tipo de objeto (por ejemplo, distintos cálculos según la subclase), un método de instancia o una interfaz polimórfica será una mejor opción.

 **Mito:** "Las clases helper sustituyen la orientación a objetos."

 **Realidad:** Falso. Las helpers complementan la arquitectura, pero no reemplazan la encapsulación ni la herencia. Su función es **apoyar**, no estructurar la lógica principal del negocio.

Resumen final

- **Métodos estáticos:** pertenecen a la clase, no a la instancia.
- **Uso ideal:** funciones puras y sin estado (ej. cálculos, conversiones, validaciones).
- **Clases helper:** agrupan métodos comunes y mejoran la reutilización del código.
- **Ubicación recomendada:** paquete util o helpers.
- **Limitaciones:** no pueden sobrescribirse ni acceder a variables de instancia.
- **Herramientas clave:** JUnit 5, SonarLint, IntelliJ Code Analyzer.
- Caso real: Spotify optimiza su arquitectura con helpers estáticos compartidos.



Sesión 18: Buenas prácticas de documentación y contratos (Javadoc)

En el desarrollo de software profesional, **la documentación no es un lujo, es una necesidad**. Un código sin comentarios claros, sin explicación de su propósito o sin guía de uso, es como un mapa sin leyenda: aunque el camino esté trazado, resulta difícil de seguir. En equipos grandes, donde diferentes personas mantienen y amplían un mismo proyecto a lo largo del tiempo, la documentación marca la diferencia entre la eficiencia y el caos.

En **Java**, el estándar de documentación es **Javadoc**, una herramienta oficial incluida en el JDK que permite generar documentación en formato **HTML directamente a partir del código fuente**. Su gran ventaja es que **el documento y el código nacen del mismo lugar**, evitando inconsistencias entre lo que está escrito y lo que realmente hace el programa.

Los comentarios de Javadoc utilizan un formato especial: comienzan con `/**` y terminan con `*/`. Dentro de ese bloque, se incluyen descripciones textuales y etiquetas con prefijo `@` que describen elementos específicos. Por ejemplo:

```
/**  
 * Calcula el área de un círculo.  
 * @param r radio del círculo  
 * @return área del círculo  
 */  
  
public static double areaCirculo(double r) {  
    return Math.PI * r * r;  
}
```

Este pequeño fragmento tiene más valor de lo que parece. Con unas pocas líneas, cualquier desarrollador puede entender **qué hace el método, qué necesita y qué devuelve**, sin necesidad de leer la implementación interna. Cuando el proyecto crece y hay cientos de clases y métodos, esta información se vuelve esencial.

El enfoque de Javadoc sigue una lógica de **contratos: cada método debe dejar claro qué espera recibir (entradas), qué ofrece (salida) y bajo qué condiciones (excepciones). Así, la documentación actúa como un **contrato formal** entre quien desarrolla el código y quien lo utiliza.

@param

describe los parámetros de entrada del método.

@return

indica qué devuelve el método.

@throws

documenta las excepciones que el método puede lanzar.

@author

(opcional) identifica al autor del código.

@version

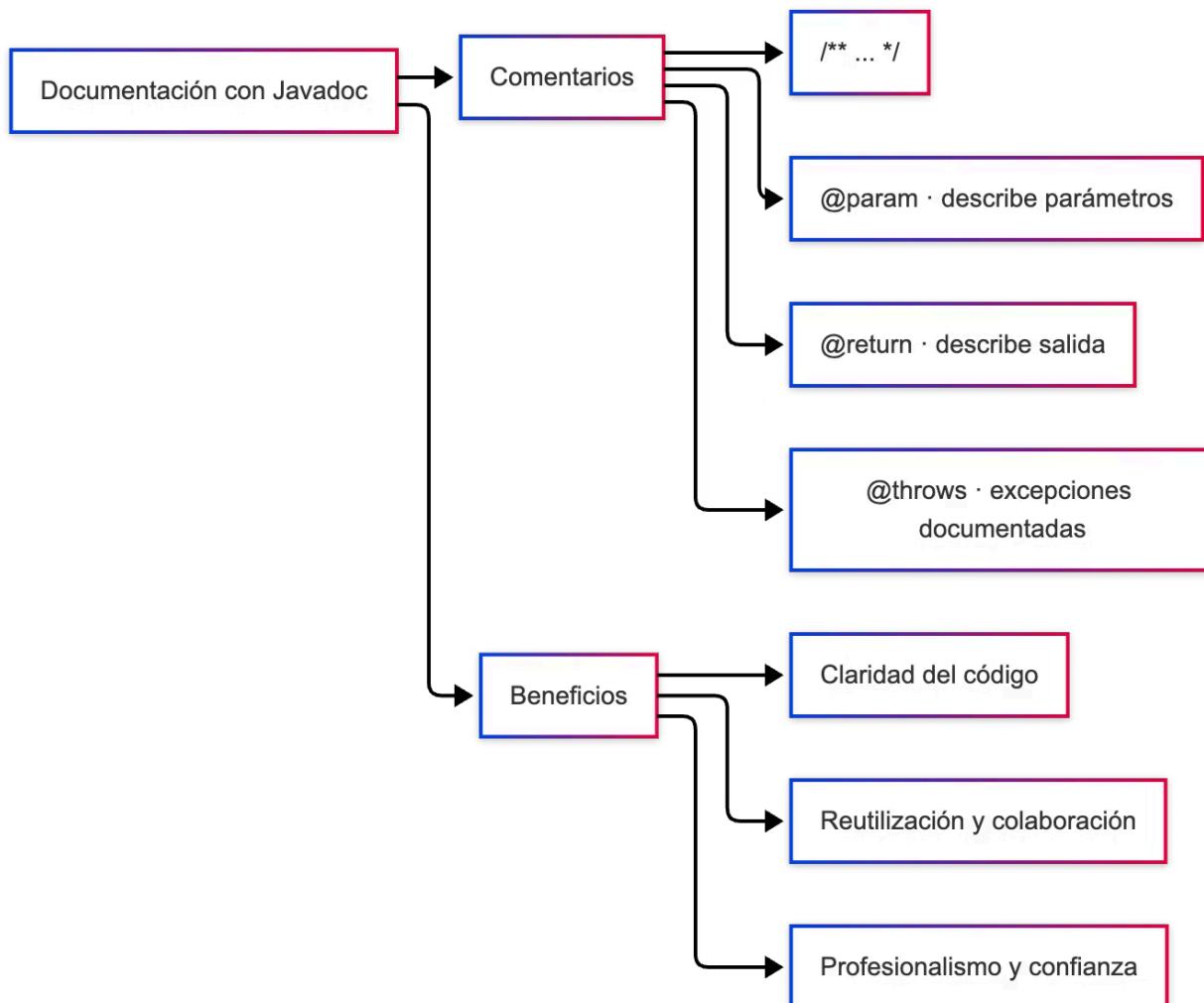
señala la versión o fecha de revisión.

El uso constante de estas etiquetas genera una documentación homogénea y navegable. Cuando se ejecuta el comando javadoc, el sistema transforma estos comentarios en **páginas web interconectadas** que permiten consultar clases, métodos y descripciones como si se tratara de una enciclopedia del proyecto.

La **documentación profesional** no se limita a describir qué hace el código, sino también **por qué** se ha diseñado de cierta forma. Incluir notas sobre decisiones técnicas o advertencias sobre limitaciones ayuda a los futuros desarrolladores (o incluso a ti mismo, meses después) a comprender la lógica detrás de cada implementación.

En resumen, **documentar es invertir en el futuro del proyecto**: ahorra tiempo, previene errores y mejora la comunicación dentro del equipo. Es una de las prácticas que diferencian a un programador aficionado de un desarrollador profesional.

Esquema visual



Explicación detallada del esquema:

Comentarios

Agrupa los elementos técnicos:

- El formato `/** ... */` indica que el comentario será procesado por la herramienta Javadoc.
- Las etiquetas `@param`, `@return` y `@throws` definen la estructura del contrato del método.

Beneficios

Destaca los resultados de aplicar correctamente Javadoc:

- **Claridad:** el código se entiende sin necesidad de leerlo completo.
- **Reutilización:** otros desarrolladores pueden usar las clases sin conocer su implementación.
- **Profesionalismo:** un proyecto bien documentado transmite calidad y confianza.

Visualmente, el esquema resume la idea de que **documentar no es escribir por escribir, sino comunicar para construir**.



Caso de estudio: La API de Java – el ejemplo de documentación perfecta

Contexto

Desde su primera versión, Oracle (y antes Sun Microsystems) ha mantenido toda la **API oficial de Java** documentada with Javadoc. Cada clase, interfaz, método o constante está descrita con detalle, permitiendo a millones de desarrolladores consultar su uso en línea sin necesidad de acceder al código fuente.

Estrategia

La documentación de la API de Java sigue una estructura y estilo consistentes que se han convertido en estándar de la industria. Por ejemplo, la clase `java.util.ArrayList` está documentada de la siguiente forma:

```
/**  
 * Resizable-array implementation of the List interface. Implements all optional  
 * list operations, and permits all elements, including null.  
 *  
 * @param<T> the type of elements in this list  
 * @author Josh Bloch  
 * @see Collection  
 * @since 1.2  
 */  
  
public class ArrayList extends AbstractList implements List, RandomAccess, Cloneable,  
    java.io.Serializable
```

Cada etiqueta cumple una función específica:

@param

define el tipo genérico que puede almacenar.

@see

crea enlaces a clases relacionadas.

@since

indica desde qué versión existe.

Gracias a esta coherencia, cualquier desarrollador puede abrir la **documentación oficial de Java** y entender inmediatamente qué hace cada clase y cómo se usa. De hecho, esta documentación es una de las más consultadas del mundo tecnológico: se estima que la API de Java recibe **millones de visitas diarias**.

Resultados e impacto

- La comunidad global de Java mantiene un estándar común de escritura y documentación.
- Los IDEs (como IntelliJ o Eclipse) integran la documentación Javadoc, mostrando la descripción de los métodos al pasar el ratón sobre ellos.
- Los frameworks y bibliotecas externas (Spring, Hibernate, Apache Commons) adoptaron el mismo formato, asegurando una experiencia uniforme.

Conclusión del caso: El ejemplo de Oracle demuestra que **una buena documentación no solo facilita el aprendizaje, sino que impulsa todo un ecosistema de desarrollo**. La API de Java no sería tan exitosa si no estuviera acompañada por una documentación clara, navegable y actualizada.

Herramientas y consejo

Consejos prácticos para un Javadoc profesional:

Documenta todo lo que sea público

Cada clase, método o atributo accesible desde fuera de su paquete debe tener su comentario. Si un compañero va a usar tu código, debe entender su propósito sin leer su implementación.

Usa las etiquetas correctamente

- `@param nombre` → Explica el significado y las condiciones del parámetro.
- `@return` → Describe qué devuelve y bajo qué circunstancias.
- `@throws` → Indica las excepciones posibles y cuándo se lanzan. Ejemplo:

```
/**  
 * Divide dos números.  
 * @param a dividendo  
 * @param b divisor (no puede ser 0)  
 * @return resultado de la división  
 * @throws ArithmeticException si el divisor es cero  
 */
```

Sé breve y claro

Evita frases extensas o redundantes. Un buen Javadoc se lee rápido y comunica mucho.

Mantén la documentación actualizada

Una documentación desactualizada es peor que ninguna. Cada vez que cambies un método, revisa sus comentarios.

Genera la documentación con el comando javadoc

Desde la consola, ejecuta:

```
javadoc -d doc *.java
```

Esto crea una carpeta doc con archivos HTML listos para navegar.

Describe el propósito, no la implementación

No escribas "suma dos números con el operador +". Es evidente. Mejor explica *por qué* el método existe o en qué contexto se usa.

Herramientas que facilitan la documentación:

IntelliJ IDEA – Javadoc Generator

Inserta automáticamente plantillas de comentarios Javadoc para métodos y clases.

Eclipse Doc Assistant

Genera secciones de documentación con etiquetas preconfiguradas y validación de parámetros.

Checkstyle

Analiza el código y verifica que todas las clases y métodos públicos estén documentados. Permite establecer políticas de estilo corporativo.

SonarQube

Evalúa la calidad del código, incluyendo el nivel de documentación, la cobertura de etiquetas y la legibilidad.

MkDocs o Jekyll (para documentación extendida)

Si el proyecto requiere documentación complementaria (guías de uso, manuales técnicos), estas herramientas pueden combinarse con el contenido Javadoc generado.

Consejo profesional:

Integra la generación de Javadoc en tu pipeline de integración continua (CI/CD). Así, cada nueva versión del proyecto incluirá su documentación actualizada automáticamente.

Mitos y realidades

 **Mito:** "La documentación retrasa el desarrollo."

 **Realidad:** En apariencia puede llevar tiempo, pero a medio plazo **reduce errores, acelera la incorporación de nuevos miembros al equipo y simplifica el mantenimiento.** Es mucho más costoso descifrar código sin comentarios que escribirlos correctamente desde el inicio.

 **Mito:** "El código bien escrito se explica solo."

 **Realidad:** Aunque el código limpio ayuda, no siempre revela el contexto, las decisiones de diseño ni las limitaciones. La documentación complementa al código, no lo sustituye.

Resumen final

- **Javadoc** genera documentación HTML directamente desde el código fuente.
- Los comentarios comienzan con `/**` y terminan con `*/`.
- Etiquetas principales: `@param`, `@return`, `@throws`.
- Se genera con el comando `javadoc *.java`.
- Mejora la comunicación, el mantenimiento y la profesionalidad del proyecto.
- Herramientas clave: IntelliJ Javadoc Generator, Eclipse Doc Assistant, Checkstyle.
- Caso de referencia: la API oficial de Java documentada por Oracle.