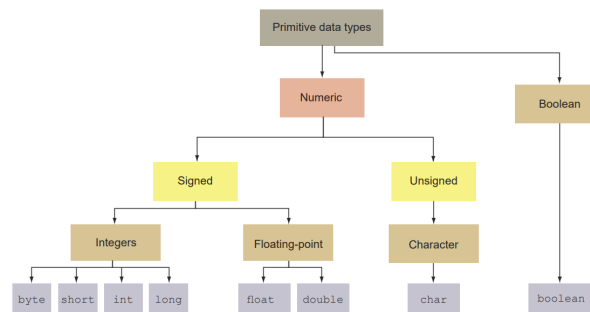


Working with Java data types

Variables primitivas

Se definen como el tipo de dato más simple en el lenguaje de programación.

En java tenemos los siguientes primitivos → **char - byte - short - int - long - float - double - boolean**



Boolean

Una variable `boolean` puede almacenar uno de los dos valores lógicos: `true` ó `false`.

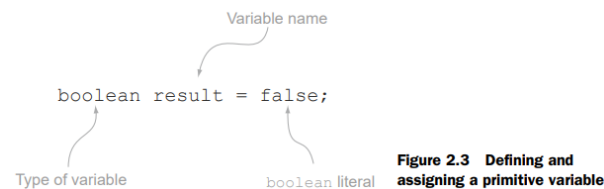


Figure 2.3 Defining and assigning a primitive variable

Los únicos **literal values** posibles en un `boolean` son `true` y `false`.

Numéricos con signo

Se divide en dos categorías más → Enteros y decimales.

Enteros → byte - int - short - long

Cada uno de estos tipos puede almacenar un rango de valores diferentes. Los beneficios son menor espacio en memoria y rapidez en procesamiento.

Table 2.4 Ranges of values stored by the signed numeric Java primitive data types

Data type	Size	Range of values
byte	8 bits	-128 to 127, inclusive
short	16 bits	-32,768 to 32,767, inclusive
int	32 bits	-2,147,483,648 to 2,147,483,647, inclusive
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807, inclusive

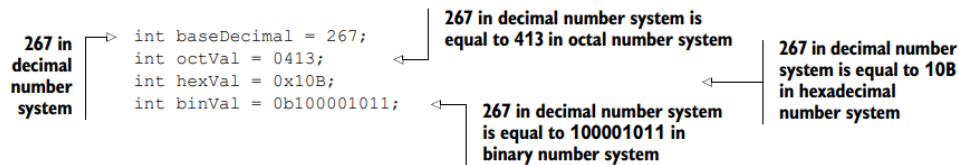
El valor por defecto de un número entero es `int`

A continuación de muestra la manera de definir variables de estos tipos:

```
byte num = 100;
short sum = 1240;
int total = 48764;
long population = 214748368;
```

Es posible meter un `int` en un `long`, únicamente hay que considerar que cuando se manejen `long` hay que agregar `L` ó `l` al final del número.

Los enteros pueden ser expresados de manera binaria, decimal, octal y hexadecimal:



- Octal → Se agrega el prefijo `0`
- Binario → Se utiliza el prefijo `0B` ó `0b`
- Hexadecimal → `0X` ó `0x`

A partir de java 7 se introducen los underscores como parte de los `literal values` para poder crear grupos de números y hacer legible el número.

```
long baseDecimal = 100_267_760;
long octVal = 04_13;
long hexVal = 0x10_BA_75;
long binVal = 0b1_0000_10_11;
```

More-readable literal values in binary, decimal, octal, and hexadecimal that use underscores to group digits and letters

Reglas para recordar

- Puedes colocar `_` después del prefijo `0` en los octales → `0_89`
- No se puede iniciar ni finalizar con `_` → `_239` | `239_`
- No puedes colocar `_` después o entre de los prefijos empleado para binarios y hexadecimales → `0b_10101` | `0_B1010` | `0_xAAAD` | `0X_3AB`
- No puedes colocar `_` antes de `L` o `f` para `long` y `float` → `1232323_L`
- No se pueden usar `_` en posiciones donde se esperen dígitos en tipo `String` o se obtendrá una excepción en tiempo de ejecución → `Integer.parseInt("45_98");`

Twist in the Tale

Determina la salida del siguiente código:

```
class TwistInTaleNumberSystems {
    public static void main (String args[]) {
        int baseDecimal = 267;
        int octVal = 0413;
        int hexVal = 0x10B;
        int binVal = 0b1_0000_1011;
```

```

        System.out.println (baseDecimal + octVal);
        System.out.println (hexVal + binVal);
    }
}

```

Los números se encuentran correctamente escritos.

La primera línea imprime: 267 + 267 = 534

La segunda línea imprime: 267 + 267 = 534

Determine which of these does this job correctly:

```

long var1 = 0_100_267_760; //Entra en el rango de int por lo que no es necesario utilizar
la L- 2^32 Aprox 4x10^9
long var2 = 0_x_4_13; //representación hexadecimal, pero no puede llevar _ entre 0x
long var3 = 0b_x10_BA_75; // Mal, o es binario o hexa
long var4 = 0b_10000_10_11; //No se puede utilizar _ despues de los prefijos 0b
long var5 = 0xa10_AG_75; //G no es un número
long var6 = 0x1_0000_10; // Bien
long var7 = 100__12_12; // Bien

```

Decimales → float - double

En java se utiliza `float` y `double` para almacenar valores decimales.

`float` requiere menos espacio `double` ya que puede almacenar un rango menor de números.

`float` es menos preciso que `double`

`float` y `double` no puede representar adecuadamente algunos números incluso si están en rango.

`float` tiene tamaño de 32 bits

`double` tiene tamaño de 64 bits

Los `float` utilizan el sufijo `F` ó `f` mientras se inicializan la variables.

El valor por defecto de una `decimal literal` es `double` .

Es posible asignar una `decimal literal` en notación científica → `double inclination = 1.2017e2;`

Es posible agregar el sufijo `D` ó `d` para especificar un valor `double` , aunque seria redundante porque un decimal ya es `double` por defecto.

A partir de Java 7 se pueden utilizar `_` con los valores de punto flotante. Las reglas generalmente son las mismas que para los valores enteros.

- No puede colocar `_` antes de `F` `f` `D` ó `d`
- No puede colocar `_` adyacente al punto decimal.

Numéricos sin signo

Character

`char` es un entero sin signo con capacidad de 16 bits y almacena un caracter unicode.

Para la designación se utilizan comillas simples → `char c1 = 'D'`

No se pueden usar comillas doble o habrá falla al compilar.

Internamente Java almacena los `char` como enteros sin signo, por lo que otra forma de definir es `char c1 = 122;`

Otra forma de definir es a través del valor unicode y con comillas simples → `char c2 = '\u0011A'`

Únicamente se pueden castear valores compatibles.

Cuando se hace un downcasting de valores se omiten los bits extras.

Es posible almacenar un negativo en `char` utilizando casting, sin embargo se esperan valores inesperados debido a que el signo de almacena como un bit de valor más.

Identificadores

Son el nombre de paquetes, clases, interfaces, métodos y variables.

Identificadores válidos

- Longitud ilimitada
- Iniciar con una letra de `a-z` upper o lowercase, un `_` y `currency signs`
- Puede usar dígitos pero no en la posición de inicio
- Puede usar `_` en cualquier posición
- Puede usar un `currency signs` en cualquier posición → `¤, $, £, €, ¥, and others`

Identificadores no válidos

- Utilizar palabras reservadas
- Usar caracteres especiales → `!, @, #, %, ^, &, *, (,), ', :, ;, [, /, \, -, }`
- Iniciar con números

Table 2.8 Java keywords and reserved words that can't be used as names for Java variables

<code>abstract</code>	<code>default</code>	<code>goto</code>	<code>package</code>	<code>this</code>
<code>assert</code>	<code>do</code>	<code>if</code>	<code>private</code>	<code>throw</code>
<code>boolean</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throws</code>
<code>break</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>transient</code>
<code>byte</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>true</code>
<code>case</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>catch</code>	<code>false</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>char</code>	<code>final</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>class</code>	<code>finally</code>	<code>native</code>	<code>super</code>	<code>while</code>
<code>const</code>	<code>float</code>	<code>new</code>	<code>switch</code>	
<code>continue</code>	<code>for</code>	<code>null</code>	<code>synchronized</code>	

Variables de referencia para objetos

Las variable en java se diferencian en dos tipos → `primitive` y `reference`

Los objetos son instancias de clase y las variables de referencia apuntan a la dirección en memoria del objeto al que es referido.

Las variables de referencia no asignadas son `null`

Un objeto que no es referido por alguna variables es elegible por el recolector de basura (removerlo de la memoria)

Un objeto puede ser referido múltiples veces por variables.

Diferencias entre variables primitivas y de referencia

La principal diferencia es que las variables primitivas almacenan directamente el valor primitivo, mientras que las variables de referencia almacenan la referencia del objeto al que se refiere.

```
int a = 77;
Person person = new Person();
```

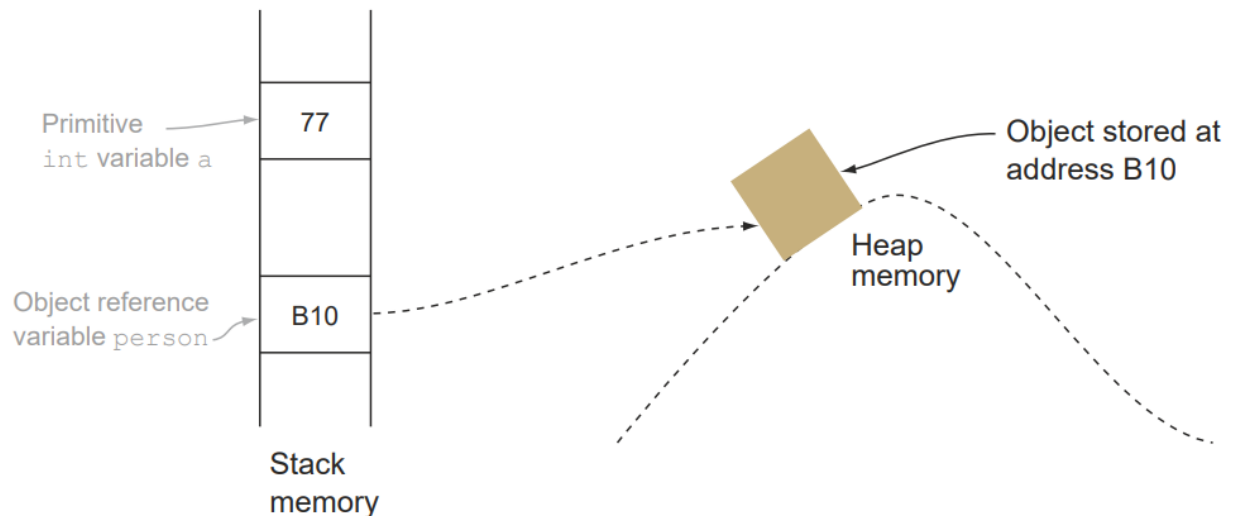


Figure 2.13 Primitive variables store the actual values, whereas object reference variables store the addresses of the objects they refer to.

- Para comparar variables de referencia se utiliza el operador `==`. Para comparar objetos, se utiliza el método `equals()`.
- Para comparar primitivos se utiliza `==`.
- El `literal value` para variables de referencia por defecto es `null`, mientras que para `boolean -> false`, `integers -> 0`, `decimal -> 0.0` y `char -> \u0000`
- Los primitivos no son marcados por el recolector de basura

Operadores

Table 2.9 Operator types and the relevant operators

Operator type	Operators	Purpose
Assignment	=, +=, -=, *=, /=	Assign value to a variable
Arithmetic	+, -, *, /, %, ++, --	Add, subtract, multiply, divide, and modulus primitives
Relational	<, <=, >, >=, ==, !=	Compare primitives
Logical	!, &&,	Apply NOT, AND, and OR logic to primitives

Asignación

El operador de asignación `=` se utiliza para inicializar variables o reasignar su valor.

```
a -= b is equal to a = a - b
a += b is equal to a = a + b
a *= b is equal to a = a * b
a /= b is equal to a = a / b
a %= b is equal to a = a % b
```

Una variable de mayor rango de capacidad no puede ser asignada a una de menor rango, no se compilará. Por ejemplo asignar un `long` a un `int`. Incluso si el valor sí cabe.

La única manera de asignar una variable de rango mayor a una de menor es a través de un casting explícito. En este caso los bits sobrantes se ignoran.

Se pueden asignar múltiples valores en una sola línea.

```
int a = 7, b = 10, c = 8;
a = b = c;
System.out.println(a);
```

Prints 8

1 Assignment starts from right; the value of c is assigned to b and the value of b is assigned to a

En este caso la asignación se realiza de derecha a izquierda.

Twist in the Tale

Examine the following code initializations and select the incorrect answers:

```
public class Foo {
    public static void main (String args[]) {
        boolean b1, b2, b3, b4, b5, b6; // line 1
        b1 = b2 = b3 = true; // line 2
        b4 = 0; // line 3
        b5 = 'false'; // line 4
        b6 = yes; // line 5
    }
}
```

- The code on line 1 will fail to compile.

Incorrecto, Puedes definir múltiples variables en una sola línea separado por coma.

- Can't initialize multiple variables like the code on line 2.

Incorrecto, Se puede inicializar múltiples variables respetando el tipo

- The code on line 3 is correct.

Es incorrecto, boolean únicamente soporta `true` `false`

- Can't assign 'false' to a boolean variable.

Efectivamente, no se puede asignar otra cosa que no sea `true` `false`

- The code on line 5 is correct.

Es incorrecto, no se puede asignar otra cosa que no sea `true` `false`

Aritméticos

Table 2.10 Use of arithmetic operators with examples

Operator	Purpose	Usage	Answer
+	Addition	12 + 10	22
-	Subtraction	19 - 29	-10
*	Multiplication	101 * 45	4545
/	Division (quotient)	10 / 6 10.0 / 6.0	1 1.6666666666666667
%	Modulus (remainder in division)	10 % 6 10.0 % 6.0	4 4.0
++	Unary increment operator; increments value by 1	++var or var++	11 (assuming value of var is 10)
--	Unary decrement operator; decrements value by 1	--var or var--	9 (assuming value of var is 10)

- El operador `unary` no se puede utilizar sobre `literal values`
- Para los `char` el operador de adición `+` suma su valor decimal correspondiente al ASCII.
- Los `char` pueden utilizar todos los operadores aritméticos.
-

Ampliación implícita de tipos de datos en una operación aritmética

Cuando se realizan operaciones sobre los numéricos `byte`, `short`, `char`, los valores son ampliados a `int`.

En caso de que la operación incluya un `long`, todos los numéricos serán ampliados a `long`.

Por ejemplo, no se puede asignar la suma de dos `byte` a un `short`.


```
byte age1 = 10;
byte age2 = 20;
short sum = age1 + age2;
```

← **Fails to compile**

Para el caso de decimales, todas las operaciones son ampliadas a `double`.

But if you modify the preceding example and define variables `age1` and `age2` as `final` variables, then the compiler *is assured* that their sum, value 30, can be assigned to a variable of type `short`, without any loss of precision. In this case, the compiler is good to assign the sum of `age1` and `age2` to `sum`. Here's the modified code:

```
final byte age1 = 10;
final byte age2 = 20;
short sum = age1 + age2;
```



Compiles successfully

Incremento y decremento unitario

Los operadores `--` y `++` son los operadores unary. Trabajan con un único operando. Se utilizan para incrementar o decrementar las variables en 1.

Los operadores Unary pueden aparecer como pre \rightarrow `++i` o pos \rightarrow `i--`.

Cuando únicamente se realiza esta única operación, el pre y el pos actúan de la misma manera.

Cuando el operador se utiliza es una expresión, el orden si importa, para el pre decimos \rightarrow Incrementa y después utiliza, mientras que para el pos decimos \rightarrow utiliza y después incrementa.

En el caso de múltiples incrementos en una sola operación, se ejecuta de izquierda a derecha.

```
int a = 10;
a = a++ + a + a-- - a-- + ++a;
System.out.println(a);

a = 10 + 11 + 11 - 10 + 10;
```

Twist in the Tale 2.3

¿Qué se imprime?

```
int a = 10;
a = ++a + a + --a - --a + a++;
System.out.println (a);
```

```
a = 11 + 11 + 10 - 9 + 9// Imprime 32
```

Operadores relacionales

Los operadores relacionales se encargan de revisar una condición. Estos operadores se utilizan para determinar si un valor primitivo es igual a otro o si es mayor o menor.

Los operadores `<` `>` `<=` `>=` trabajan con todos los `numbers`, incluidos `Integers` y de punto flotante.

No se pueden comparar valores como `boolean` con `char` `int`, etc. No compilará.

Comparando igualdad de primitivos

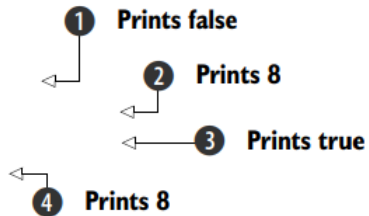
Los operadores `==` y `!=` trabajan con todos los primitivos, y su retorno es un `boolean`

Operadores lógicos

Estas expresiones retornan un valor `boolean` y evalúan la relación entre expresiones. `AND` `NOT` `OR`

Los operadores `&&`, `||` son llamados de corto circuito (short-circuit). Ya que cuando se encuentran con una condición determinada ya, se omite la evaluación del segundo operando.

```
int marks = 8;
int total = 10;
System.out.println(total < marks && ++marks > 5);
System.out.println(marks);
System.out.println(total == 10 || ++marks > 10);
System.out.println(marks);
```



1 Prints false
2 Prints 8
3 Prints true
4 Prints 8

Los operadores `&` y `|` no son de corto circuito.

Twist in the Tale 2.4

```
class TwistInTaleLogicalOperators {
    public static void main (String args[]) {
        int a = 10;
        int b = 20;
        int c = 40;
        System.out.println(a++ > 10 || ++b < 30); // true - a=11 b =21
        System.out.println(a > 90 && ++b < 30); //false - a=11 b =21
        System.out.println(!(c>20) && a==10 ); //false - a=11 b =21
        System.out.println(a >= 99 || a <= 33 && b == 10); //false - a=11 b =21
        System.out.println(a >= 99 && a <= 33 || b == 10); //false
    }
}
```

Precedencia de operadores

En caso de tener una línea con múltiples operaciones, se realizan de izquierda a derecha empezando por la mayor jerarquía mostrada a continuación:

Table 2.12 Precedence of operators

Operator	Precedence
Postfix	Expression++, expression--
Unary	++expression, --expression, +expression, -expression, !
Multiplication	* (multiply), / (divide), % (remainder)
Addition	+ (add), - (subtract)
Relational	<, >, <=, >=
Equality	==, !=
Logical AND	&&
Logical OR	
Assignment	=, +=, -=, *=, /=, %=

Se pueden utilizar parentesis para sobrescribir el orden por defecto de las operaciones.

Wrapper classes

Java define un wrapper por cada clase de primitivo. Las clases Wrapper son utilizadas para envolver los primitivos y tratarlos como objetos.

Jerarquía de los Wrapper

Todas las clase Wrapper son **inmutables**, por lo que no se permite cambio en su estado una vez inicializado.

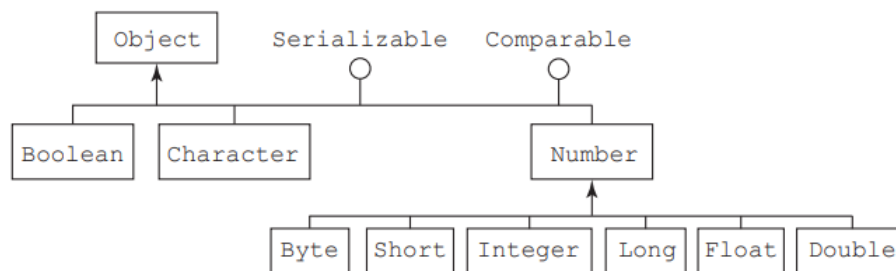


Figure 2.18 Hierarchy of wrapper classes

Creando objetos de las clases Wrapper

- Por asignación → Asignando un primitivo a una variables de la clase del Wrapper (**Autoboxing**)
- Por constructor → Utilizando la clase constructor del Wrapper
- Por métodos estáticos → Llamando métodos estáticos de la clase, por ejemplo `valueOf()`

```
Boolean bool1 = true;
Character char1 = 'a';
Byte byte1 = 10;
Double double1 = 10.98;
```

Autoboxing

```
Boolean bool2 = new Boolean(true);
Character char2 = new Character('a');
Byte byte2 = new Byte((byte)10);
Double double2 = new Double(10.98);
```

**Constructors that
accept primitive value**

```
Boolean bool4 = Boolean.valueOf(true);
Boolean bool5 = Boolean.valueOf(true);
Boolean bool6 = Boolean.valueOf("TrUE");
Double double4 = Double.valueOf(10);
```

**Using static
method valueOf()**

- Ningún Wrapper define un constructor sin argumentos.
- Todos los wrapper tienen un constructor que acepta un argumento `String`.

Recuperando el valor primitivo de un Wrapper

Todos los Wrappers definen un método `primitive Value()`, en donde *primitive* es el nombre del tipo de dato, que recupera el valor primitivo que se almacena. Este método se hereda de `Number`.

Convirtiendo un String a un valor primitivo

Para obtener el valor de un `String` en un primitivo se tiene el método estático `parse DataType ()`, donde *DataType* se refiere al tipo de retorno.

Todos los Wrapper (Excepto `Character`) definen el método que recibe un `String`.

Table 2.14 List of `parseDataType` methods in wrapper classes

Class name	Method
Boolean	<code>public static boolean parseBoolean(String s)</code>
Character	no corresponding parsing method
Byte	<code>public static byte parseByte(String s)</code>
Short	<code>public static short parseShort (String s)</code>
Integer	<code>public static int parseInt(String s)</code>
Long	<code>public static long parseLong(String s)</code>
Float	<code>public static float parseFloat(String s)</code>
Double	<code>public static double parseDouble(String s)</code>

Estos métodos arrojan la excepción `NumberFormatException` cuando se les da valores inválidos. Excepto el caso de `Boolean`, este retorna `false` cuando se pasa cualquier `String` diferente a `true`.

```

Long.parseLong("12.34");
Byte.parseByte("1234");
Boolean.parseBoolean("true");
Boolean.parseBoolean("TrUe");

```

Throws `NumberFormatException`:
12.34 isn't a valid long

Throws `NumberFormatException`:
1234 is out of range for byte

Returns boolean true

No exceptions; the String argument isn't case-sensitive

El argumento no es case-sensitive.

Diferencia entre `valueOf()` y los constructores - POOL

El Wrapper de las clases `Byte`, `Short`, `Integer` y `Long` tiene un caché para valores entre `-128` y `127`. El caché para la clase `Character` va de `0` a `127`. Entonces si un valor dentro de estos rangos es requerido, `valueOf()` retorna la referencia a un objeto ya definido, o creará uno nuevo en caso de que no haya uno ya definido.

El Wrapper `Float` y `Double` no tienen caché.

Los constructores siempre crean nuevas instancias, no recurren al caché.

Autoboxing sí recurre al caché.

Comparando objetos de las clases Wrapper

El método `equals()` siempre compara el valor primitivo almacenado por el Wrapper.

El operador `==` compara las referencias del objeto.

No se pueden comparar si no se pertenece a la misma clase. El código no compilará cuando se utiliza `==`, y cuando se use `equals()` el retorno es `false`.

Autoboxing y Unboxing

Autoboxing se refiere a la conversión automática de un valor primitivo a un objeto del correspondiente Wrapper.

Unboxing se refiere al proceso inverso.

Las clases Wrapper son inmutables, por lo que cada vez que se asigna un nuevo valor, en realidad se crea un nuevo objeto.

Hacer unboxing de variables cuya referencia es `null` provoca `NullPointerException` debido a que internamente se utiliza el método `primitive Value()`.

▼ Sample exam questions

Q1.- Dado

```

int myChar = 97;
int yourChar = 98;
System.out.print((char)myChar + (char)yourChar);
int age = 20;
System.out.print(" ");
System.out.print((float)age);

```

What is the output?

- a 195 20.0
- b 195 20

- c ab 20.0
- d ab 20
- e Compilation error
- f Runtime exception

Cuando se operan `char` se utiliza su valor ASCII.

Q2.- Which of the options are correct for the following code?

```
public class Prim { // line 1
    public static void main(String[] args) { // line 2
        char a = 'a'; // line 3
        char b = -10; // line 4
        char c = '1'; // line 5
        integer d = 1000; // line 6
        System.out.println(++a + b++ * c - d); // line 7
    } // line 8
} // line 9
```

- a Code at line 4 fails to compile.
- b Code at line 5 fails to compile.

c Code at line 6 fails to compile.

d Code at line 7 fails to compile.

No se pueden asignar negativos a `char`

No existe un tipo `integer`

Q3.- What is the output of the following code?

```
public class Foo {
    public static void main(String[] args) {
        int a = 10;
        long b = 20;
        short c = 30;
        System.out.println(++a + b++ * c); // 11 + 20 * 30 -> 11 + 600 -> 611
    }
}
```

- a 611**
- b 641
- c 930
- d 960

Q4.- Given:

```
Boolean buy = new Boolean(true);
Boolean sell = new Boolean(true);
System.out.print(buy == sell);
boolean buyPrim = buy.booleanValue();
```

```
System.out.print(!buyPrim);  
System.out.print(buy && sell);
```

What is the output?

a falsefalsefalse

b truefalsetrue

c falsetruetrue

d falsefalsetrue

e Compilation error

f Runtime exception

El constructor crea nuevas instancias de objetos, entonces la instancia es diferente.

Q5.- Which of the following options contain correct code to declare and initialize variables to store whole numbers?

a bit a = 0;

b integer a2 = 7;

c long a3 = 0x10C;

d short a4 = 0512;

e double a5 = 10; → No se almacenan enteros

f byte a7 = -0;

g long a8 = 123456789;

Q6.- . Select the options that, when inserted at // INSERT CODE HERE, will make the following code output a value of 11:

```
public class IncrementNum {  
    public static void main(String[] args) {  
        int ctr = 50;  
        // INSERT CODE HERE  
        System.out.println(ctr % 20);  
    }  
}
```

a ctr += 1;

b ctr =+ 1;

c ++ctr;

d ctr = 1;

Q7.- What is the output of the following code?

```
int a = 10;  
int b = 20;
```

```
int c = (a * (b + 2)) - 10-4 * ((2*2) - 6;  
System.out.println(c);
```

- a 218
- b 232
- c 246

d Compilation error

Q8.- What is true about the following lines of code?

```
boolean b = false;  
int i = 90;  
System.out.println(i >= b);
```

- a Code prints true
- b Code prints false
- c Code prints 90 >= false

d Compilation error

Q9.- Examine the following code and select the correct options:

```
public class Prim { // line 1  
    public static void main(String[] args) { // line 2  
        int num1 = 12; // line 3  
        float num2 = 17.8f; // line 4  
        boolean eJavaResult = true; // line 5  
        boolean returnVal = num1 >= 12 && num2 < 4.567 // line 6  
        || eJavaResult == true;  
        System.out.println(returnVal); // line 7  
    } // line 8  
} // line 9
```

- a Code prints false

- b Code prints true

c Code will print true if code on line 6 is modified to the following:

boolean returnVal = (num1 >= 12 && num2 < 4.567) || eJavaResult == true;

d Code will print true if code on line 6 is modified to the following:

boolean returnVal = num1 >= 12 && (num2 < 4.567 || eJavaResult == false);

Q10.- Given:

```
boolean myBool = false; // line 1  
int yourInt = 10; // line 2  
float hisFloat = 19.54f; // line 3  
System.out.println(hisFloat = yourInt); // line 4
```

```
System.out.println(yourInt > 10); // line 5  
System.out.println(myBool = false); // line 6
```

What is the result?

a true

true

false

b 10.0

false

false

c false

false

false

d Compilation error