**Primitives vs. References**

* **primitive types** are the basic types of data
  + byte An 8-bit (1-byte) integer value
  + short A 16-bit (2-byte) integer value
  + int A 32-bit (4-byte) integer value
  + long A 64-bit (8-byte) integer value
  + float A 32-bit (4-byte) floating-point value
  + double A 64-bit (8-byte) floating-point value
  + Boolean A true or false value
  + Char A 16-bit character using the Unicode encoding scheme
  + primitive variables store primitive values
* **reference types** are any instantiable class as well as arrays
  + String, Scanner, Random, Die, int[], String[], etc.
  + reference variables store addresses

**Assignment**

* copies the contents of RHS variable into LHS variable
  + primitives: the primitive value is copied
  + references: the address is copied
* **implications**: for references the object is not copied, it is shared (reference variables are *aliases*)

**Comparisons (e.g. ==)**

* compares the contents of the variables
  + primitives: the primitive values are compared
  + references: the addresses are compared
* **implications**: for references the contents of the objects are not compared

**Passing Parameters**

* **terminology**:
  + formal parameter: the parameter variable that is listed (along with its type) in the method declaration
  + actual parameter: the parameter that is given when the method is called
* copies the contents of actual parameter into the formal parameter (i.e., ***pass-by-value***)
  + primitives: the primitive value is copied
  + references: the address is copied
* **implications**: for references the object is not copied, it is shared (i.e., actual parameter and formal parameter are aliases)
* **primitives**: changing the formal parameter's value doesn't affect the actual parameter's value
* **references**: changing the formal parameter's address doesn't affect the actual parameter's address **but** changing the formal parameter's object does change the actual parameter's object since they refer to the same object

**Returning Values**

* returns a result to where the method was called
  + primitives: the primitive value is returned
  + references: the address is returned
* recall: local variables and parameters are destroyed when the method finishes execution
* **implications**: a locally created object can survive if it is returned or if it is stored in a data member

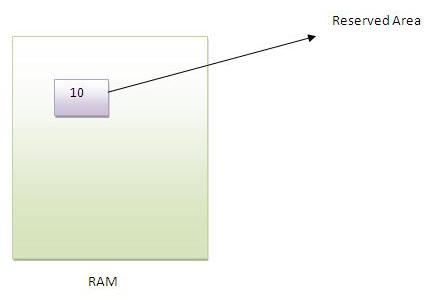
# **Variable and Datatype in Java**

1. [Variable](http://www.javatpoint.com/variable-datatype#variable)
2. [Types of Variable](http://www.javatpoint.com/variable-datatype#typesofvariable)
3. [Data Types in Java](http://www.javatpoint.com/variable-datatype#datatype)

|  |
| --- |
| In this page, we will learn about the variable and java data types. Variable is a name of memory location. There are three types of variables: local, instance and static. There are two types of datatypes in java, primitive and non-primitive. |

### Variable

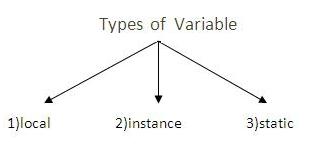
Variable is name of reserved area allocated in memory.



1. **int** data=50;//Here data is variable

### Types of Variable

|  |
| --- |
| There are three types of variables in java   * local variable * instance variable * static variable |



#### Local Variable

|  |
| --- |
| A variable that is declared inside the method is called local variable. |

#### Instance Variable

|  |
| --- |
| A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static. |

#### Static variable

|  |
| --- |
| A variable that is declared as static is called static variable. It cannot be local. |

|  |
| --- |
| We will have detailed learning of these variables in next chapters. |

#### **Example to understand the types of variables**

1. **class** A{
2. **int** data=50;//instance variable
3. **static** **int** m=100;//static variable
4. **void** method(){
5. **int** n=90;//local variable
6. }
7. }//end of class

### Data Types in Java

|  |
| --- |
| In java, there are two types of data types   * primitive data types * non-primitive data types |



|  |  |  |
| --- | --- | --- |
| **Data Type** | **Default Value** | **Default size** |
| boolean | false | 1 bit |
| char | '\u0000' | 2 byte |
| byte | 0 | 1 byte |
| short | 0 | 2 byte |
| int | 0 | 4 byte |
| long | 0L | 8 byte |
| float | 0.0f | 4 byte |
| double | 0.0d | 8 byte |

### Why char uses 2 byte in java and what is \u0000 ?

because java uses unicode system rather than ASCII code system. \u0000 is the lowest range of unicode system.To get detail about Unicode see below.

# **Unicode System**

|  |
| --- |
| Unicode is a universal international standard character encoding that is capable of representing most of the world's written languages. |

### Why java uses Unicode System?

|  |
| --- |
| Before Unicode, there were many language standards: |
| * **ASCII** (American Standard Code for Information Interchange) for the United States. * **ISO 8859-1** for Western European Language. * **KOI-8** for Russian. * **GB18030 and BIG-5** for chinese, and so on. |

|  |
| --- |
| **This caused two problems:**   1. A particular code value corresponds to different letters in the various language standards. 2. The encodings for languages with large character sets have variable length.Some common characters are encoded as single bytes, other require two or more byte. |
| To solve these problems, a new language standard was developed i.e. Unicode System. |
| In unicode, character holds 2 byte, so java also uses 2 byte for characters. |
| **lowest value:**\u0000 |
| **highest value:**\uFFFF |

# **Operators in java**

**Operator** in java is a symbol that is used to perform operations. There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.

|  |  |
| --- | --- |
| **Operators** | **Precedence** |
| postfix | expr++ expr-- |
| unary | ++expr --expr +expr -expr ~ ! |
| multiplicative | \* / % |
| additive | + - |
| shift | << >> >>> |
| relational | < > <= >= instanceof |
| equality | == != |
| bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| logical AND | && |
| logical OR | || |
| ternary | ? : |
| assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |