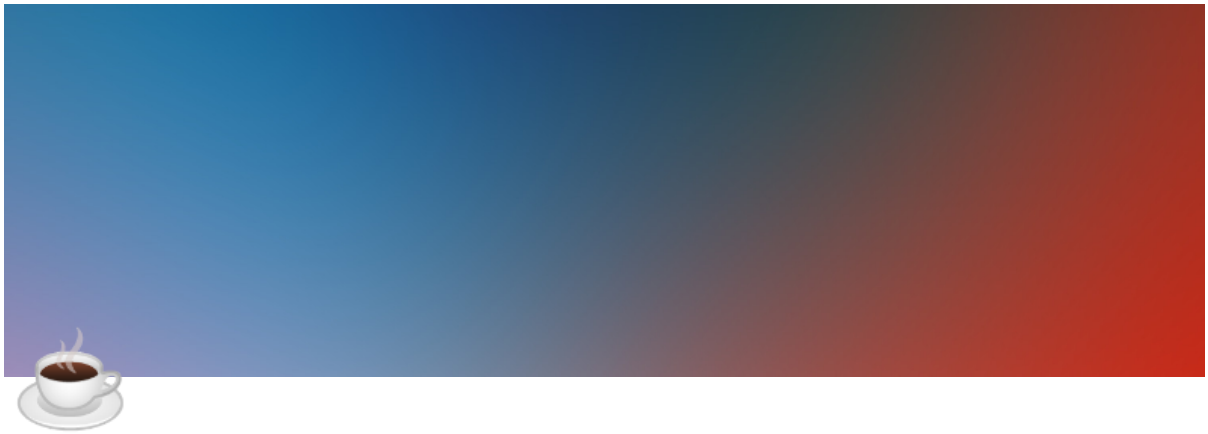


Datatypes in Java

Warm wishes to the Future Developer. The time has come when we have to analyze how Java works with the datatypes(i.e. the type of data a variable/container can store) . Well, Java is strictly and strongly typed language ever since it was born. Strongly typed does not mean that we need to type the keyboard keys very strongly. It means that the programmer must specify what type of data he/she is going to store in the containers. For example, if you make a Swimming Pool for your future house, you must decide what it will store when it is built, and in this case, a Swimming pool stores water. In the same similar way, if the compiler knows about what data is stored in a variable, Compiler can draw out quick conclusions out from the calculations.

What are Datatypes?

Datatypes: **The type of data a variable can hold is called a Datatype.**

- Datatypes allow variables to act as a container of a specific type which can store the values of the same specific type.Eg: To store liquids, you will use a container or vessel. That vessel may store different volumes of water. But all those vessel will store only liquids.
    - *Hence* ***Liquid is a type of data a vessel(variable) can hold*** *and the volume of the vessel is the range/capacity to which the variable can hold the type of data*.**Vessel = VariableVolume of Vessel = Range of DatatypeLiquid = Type of Data.**

Why do we need a Datatype?

- For Example: Let’s consider the same example given above. When you know the type of data, it becomes easier for you to choose a vessel. Similarly, when

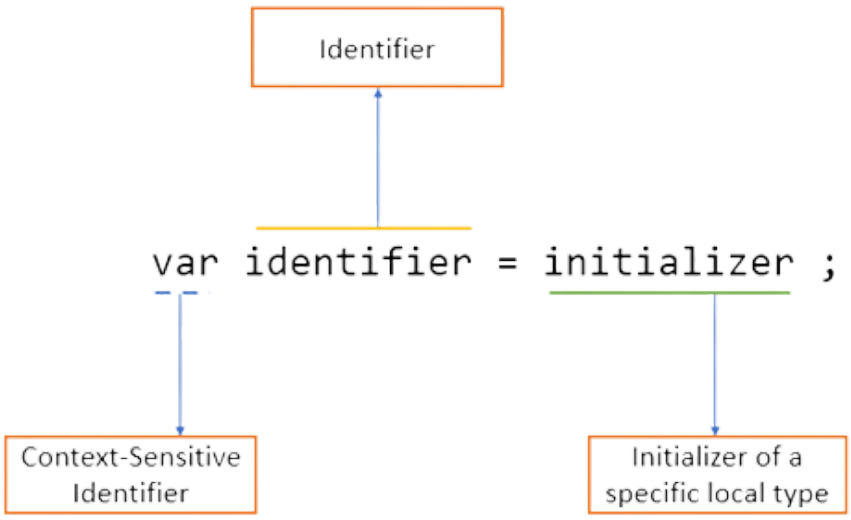
we consider this in Technological Terms, **it becomes easier for the Compiler to choose a variable of convenient size and range.**

- But in the next case, we would see that the Compiler would detect the datatype automatically and would choose the required vessel.
- It **increases the Efficiency of the Program** during Runtime. It is **because the Compiler knows how much memory is to be allocated for each variable of the given type.**
- **It would make a program robust and error free by preventing any kind of incompatible type error.**

Case Study : Local Variable Type Inferencing

- An Interesting thing is to know that with JDK 10, you can use variables without even using datatypes.
- We will very soon learn that **So as to use variables, we *Must declare the variable prior to its use and it must be initialized with a value and its initializer type.***
- But, in this case, the Compiler would automatically detect the type of initializer.
- This reduces code redundancy and helps in situations where deciding the type is difficult to make out or cannot be denoted.
- *Definition*: **Local Variable Type Inferencing is a concept where the Compiler automatically identifies/recognizes the type of data that is to be stored in the variables.**
- It is achieved by using a **context-specific identifier** namely: **var**.
- **var** is context-specific since it depends on the place of the usage in a Java Program.
- Eg: Let's Consider the Given Example: -

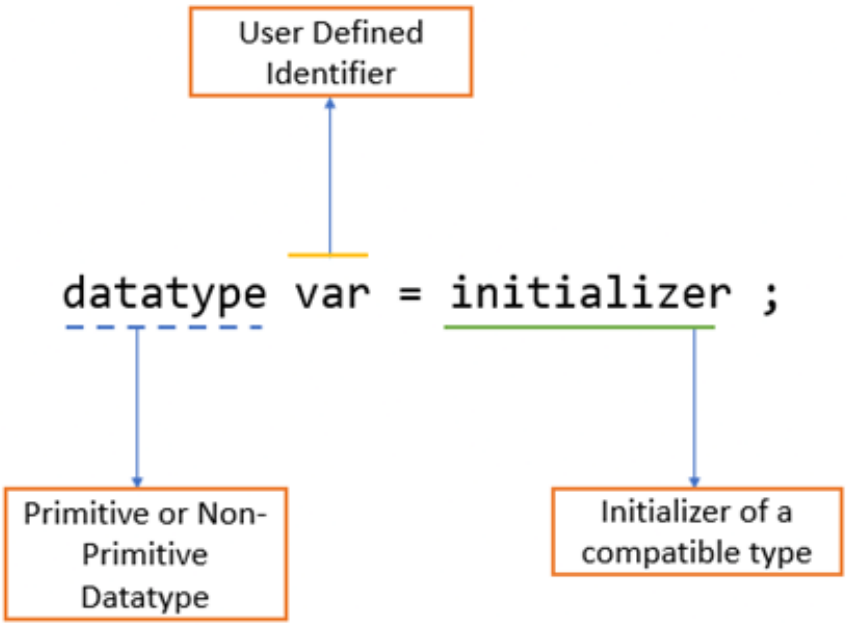
```
    public class Foo{
        public static void main(String[] args){
            var i = 10;        //var is an context specific identifier here.
            int var = 10;     //var is an context-indefinite/user-defined identifi
er here.
        }
    }
```

- 

- In the above picture, its visible that **var** acts as a **context-specific identifier**.
- **Definition**: **Context-Specific Identifier means an identifier(not a reserved word) having different functionalities when used at different places in a Java program.**
- Eg: of a program where **var** is an user-defined identifier.

```
    public class Demo2{
        public static void main(String[] args){
            int var[] = {1,23,34,2,2,1,2,3};  //Here var is an user defined identi
fier.
        }
    }
```

- This Program shown above would not show any error.
- Syntax of User-Defined **var** named Identifier.



- We would learn more about Local Variable Type Inferencing very soon in another Notes.

Is Java Statically Typed Language?

1. The Answer is ***Yes***.
2. **Reasons: -**
   - **A variable must be declared with a datatype before its use.**
   - **It must be initialized with an *initializer* before use**.
3. **Definition: Statically Typed Language is a Programming Language in which the Programmer needs to specify the type of data a variable can hold *EXPLICITLY* so that the Compiler can detect the common type errors long before the program actually executes**.

Question: Do you think that Java is still statically typed after the introduction of Local Variable Type Inferencing in JDK 10.?

Fooling around with Java Datatypes.

1. **View the Type Dissociation into Primitive and Non-Primitive**
2. In the image shown above, it is clear that there are two forms of Datatypes namely: **Primitive and Non-Primitive**.

Primitive Datatypes

- **Signed Numbers --> Signed Numbers are numbers whose range is from a negative to a positive number and these numbers contain the Most Significant Bit which denotes whether the number is negative or positive.**
- **Datatypes having a fixed range and a predefined size built for just improving the efficiency of the program that cannot be instantiated, are called Primitive Datatypes.**
- But what is **Instantiation**?
- **Instantiation means creating the objects so that its properties(variables) like hair color, skin color, etc. and behavior(methods) like eating, sleeping, etc. can be accessed.** We will learn more about Instantiation in the subsequent notes of Object Oriented Programming.
- There are some **special characteristics** of these types: -
    - **They have a fixed size.**Eg: **Visit this Link** You have measuring cylinders which consumes some area when placed on a table.
    - **They have a fixed range.**Eg: For a measuring cylinder, they can store only a fixed volume. If we try to fill more, an overflow happens. In Java, overflow of range in case of Primitive types results in a syntax error namely **integer number too large**.
    - **They store a specific datatype**Eg: You store water in an overhead tank and not fruit juice. In case of Java, if there is a type mismatch, then a syntax error namely **incompatible types**.
    - **You cannot create the object of Primitive types**EG:

```
int a = new int(20);        //Results in an error.
int a = 20;         //Works Successfully
```

- Its important to see the division of Java Datatypes in the Primitive and Non-Primitive Hierarchy. If you not yet seen it, its important to see it…

Why there are Primitive types at all in Java?

1. Answer to this, it is just for increasing the efficiency of a Java Program both by Memory and Time.

2. As already mentioned, its easier for you to choose a vessel prior to filling up the liquid in the vessel.
3. Now watch this dramatic conversation between a Compiler and a Programmer on the program given below.

```
 public class Demo3{
     public static void main(String args[]){
         int a = 20;
         double b = 90.4543;
         char ch = 'a';
    }
}
```

Programmer: Hey *Compiler*, please **compile** this source code and create a bytecode file.**Compiler**: Okay dude, there are no syntax errors and everything seems fine. I am compiling the code…Please wait a moment.**Compiler**: Hey Pal, I have compiled the code and have mentioned **how many bytes of memory should be allocated for the Primitive types you initialized in the ByteCode**.(*Bytecode or intermediate code is the highly optimized, platform independent set of instructions that is passed to the JVM to be converted to Machine Native Code.*)**Programmer**: Why do you need to tell the JVM that how much size is to be required?**Compiler**: It is necessary Pal. If I dont inform JVM about the size, it would have to search it by himself and that would take a lot of time.**Compiler**: Moreover, I cannot let you and JVM struggle for time and space.**Programmer**: Greatful to you, my friend.

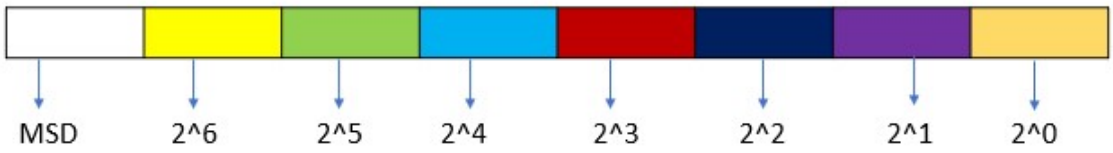
1. So, this is the dramatic reason why Primitive types have a fixed range and size.

Time to know about the Primitive Datatypes one by one.

Integral Datatypes

byte

1. **Size --> 1 Byte or 8 bits.**
2. **Range --> -128 to 127.**
3. Generally used in case of **transferring Strings across Networks and writing files with Character and Byte Streams**.
4. **MAX_VALUE : 127**

5. **MIN_VALUE : -128**
6. 

7. The **MSD** stands for **Most Significant Bit**. This stores either 0(negative) or 1(positive). **In Signed Integers, this MSD signifies whether the number stored is positive(zero or more than zero) or negative(less than zero).**
8. If you add **2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0**, you would get the sum as **127**.

Operations on a *byte* datatype

```
public class Demo4{
  public static void main(String[] args){
    double a = 20.52; byte b = a; //Possible Loss of Precision.
  }
}
```

```
public class Demo5{
         public static void main(String[] args){
             String str = "Krish Jaiswal";
             byte b = str;    // Incompatible Types. found: java.lang.String ; require
d: byte
         }
     }
```

Infamous Error in case of Datatypes are :-

1. **Possible Loss of Precision**: It is caused due ***to possible loss of some value during assignment to a variable having a different type***.
2. **Incompatible Types**: It is caused when two datatypes, which are ***not compatible and hold no relation with each other are assigned or explicitly casted.***

---

short

1. **Size --> 2 Byte or 16 bits.**
2. **Range --> -32768 to 32767.**
3. Was **previously used for Intel 8086 16 bit Processors which existed during late 1990s and early 2000s since it provided efficiency there too.**
4. **MAX_VALUE : 32767**
5. **MIN_VALUE : -32768**
6. The **MSD** stands for **Most Significant Bit**. This stores either 0(negative) or 1(positive). **In Signed Integers, this MSD signifies whether the number stored is positive(zero or more than zero) or negative(less than zero).**
7. If you add **2^15 + 2^14 + 2^13 + 2^12 + 2^11 + 2^10 + 2^9** + …, you would get the sum as **32767**.

Operations on a *short* datatype

```
public class Demo6{
         public static void main(String[] args){
             double a = 20.52;
             short b = a; //Possible Loss of Precision.
         }
     }
```

```
     public class Demo7{
         public static void main(String[] args){
             String str = "Krish Jaiswal";
             short b = str;    // Incompatible Types. found: java.lang.String ; required:
         }
     }
```

---

int

1. **Size --> 4 Bytes or 32 bits.**
2. **Range --> -2147483648 to 2147483647.**
3. It is the **default integral datatype. It is frequently used. It has a wide range of uses.**
4. **MAX_VALUE : 2147483647**
5. **MIN_VALUE : -2147483648**

6. If you add **2^31 + 2^30 + 2^29 + 2^28 + 2^27 + 2^26 + 2^25** + …, you would get the sum as **2147483647**.

Operations on a *int* datatype

```
class Demo8{
         public static void main(String[] args){
             int b = 10l; //Possible Loss of Precision since "l" denotes a long datatype.
         }
     }

class Demo9{
         public static void main(String[] args){
             String str = "Krish Jaiswal";
             int b = str;    // Incompatible Types. found: java.lang.String ; required: i
         }
     }


  class Demo10{
         public static void main(String[] args){
             int a = 20;
             int b = a*4+1;
             System.out.println(b);
         }
     }
```

Note : Run these programs in different files to understand them better. Since they are independent programs, they wont affect or cause any kind of error.

long

1. **Size --> 8 Bytes or 64 bits.**
2. **Range --> -9_223_372_036_854_775_808 to 9_223_372_036_854_775_807.**
3. It is a signed 64 bit type used for storing excessive large or small numbers that may surpass the range of **int** datatype.
4. **MAX_VALUE : 9_223_372_036_854_775_807**
5. **MIN_VALUE : -9_223_372_036_854_775_808**
6. If you add **2^63 + 2^62 + 2^61 + 2^60 + 2^59 + 2^58 + 2^57** + …, you would get the sum as **9_223_372_036_854_775_807**.

Operations on *long* datatype.

```
class Demo11{
         public static void main(String[] args){
             String str = "Krish Jaiswal";
             long b = str;    // Incompatible Types. found: java.lang.String ; required:
         }
     }

 class Demo12{
         public static void main(String[] args){
             long var1 = 104 * 203 ;
             long var2 = 2147483648 ; //Gives a compile time error: integer number too la
             /*
             * A long datatype has a default value as 0l.
             * Here 'l' denotes that the number is of long type.
             * Since all numbers are by default of 'int' type, Compiler gives an error if
             */

             long var3 = 2147483648l ; // No Error.
         }
     }
```

Floating-Point Types

1. Java implements the standards of IEEE 754 set of floating point types and operators.
2. Floating Point means **real numbers**.Eg: There are 2 consecutive numbers. Let's say 20 and 21. So the number between 20 and 21 can be 20.302, 20.1, 20.2, etc.

float

1. **Size --> 4 Bytes or 32 bits.**
2. **Range --> 1.4e-045 to 3.4e+38.**
3. It specifies a *single-precision* value that uses 32 bits of storage. Single precision are useful on some processors and hence is used in storing floating point values.

Operations on *float* datatype

```
public class Demo13{
         public static void main(String[] args){
             float f = 2.13421f;
             /*
```

```
             * Since all the floating point numbers have 'double' type as default type, w
             * This is done by adding 'f' to the number.
             */

             float f2 = 32.3212123 ; //Compile Time Error occurs: Possible Loss of Precis
         }
     }
```

double

1. **Size --> 8 Bytes or 64 bits.**
2. **Range --> 4.9e-324 to 1.8e+308 .**
3. It specifies a Double Precision that uses 64 bit of storage. It is the default datatype for all floating datatypes.

Operations on *double* datatype

```
public class Demo14{
         public static void main(String[] args){
             double d = 2.48213121;
         }
     }
```

Character

char

1. This datatype is **used to store characters.**
2. It uses **Unicode standard that is a fully international character set of 2^16 characters containing Arabic, Chinese, Hebrew, etc. totalling to 65,535 characters.**
3. It is **not a signed datatype. It means that it cannot have a negative value**.
4. Unicode system is combination of other smaller character sets like **ISO-Latin1** and **ASCII.**
5. Characters are **internally represented as integral numbers.**Eg: Try executing this code:-Output: -

```
    public class Demo15{
        public static void main(String[] args){
            char ch = 'a';
            int asc = ch;    //Compiler internally represents a character as an in
teger number and hence assigning a character to an integer type variable gives no
 errors at all.
            System.out.println("Integer Value of "+ch+" is : "+asc);
        }
    }
```

```
    Integer Value of a is : 97
```

6. We would be understanding **Implicit Type Casting** in some other lectures.

Boolean

boolean

1. Does your toothpaste have salt in it?
2. Are you 54 years old?
3. *boolean* is a datatype which helps us to **classify a condition in *true* and *false*** .
4. So, imagine this situation:-
    - Person 1: Is it sunny Outside?
    - Person 2: No (false).
    - Person 2: Should we carry umbrella?
    - Person 1: No (false).
5. These things are represented by **boolean** datatype.
6. All **Relational and Logical conditions are either in true or in false, which means they have boolean as the only type.**
7. Interestingly, it **cannot be type casted (either implicitly or explicitly) to any other datatype.** Doing that would give a Compile Time Error : **incompatible types**.

Operations on *boolean* datatype

```
public class Demo16{
         public static void main(String[] args){
             boolean b = true;
             boolean x = false;

             int a = 20;
             boolean result = (a>20)?true:false;    //Ternary Operations would be cove
red in subsequent lectures.
             int val = 1;
             boolean wrong = val; //Compile Time Error. Incompatible types.
         }
     }
```

How Prepared you Are?

1. What would be the first line to generate an error in this program.: -

```
    public class Test2{
        public static void main(String[] args){
            int a = 10.34;         // 1
            double b = 45.302f;    //2
            char ch = 97;        //3;
        }
    }
```

2. Why Java makes use of primitive datatypes?
3. Predict the Output:-

```
    public class Test3{
        public static void main(String[] args){
            int g = 'a';
            System.out.println(g);
        }
    }
```

4. Predict the Output: -

```
    public class Test4{
        public static void main(String[] args){
            double 2ft = 50.32f;
        }
    }
```