Type Checking

Type checking depends on the fact that every variable declaration gives the type checking of the variable, and the header of every method and constructor defines its *signature*: the types of its arguments and results (and also the types of any exceptions it throws).

This information allows the compiler to deduce an **apparent type** for any expression and this deduction then allows it to determine the legality of an assignment

For Java is *a strongly types* language, which means that the Java compiler checks the code to ensure that every assignment and every call is type correct. If a type error is discovered, compilation fails with an error message.

example consider

```
int y= 7;
int z= 3;
int x= num.gcd (z,y); // discuss that
```

When the compiler processes the call to class ***Num.gcd***, it knows that the ***Num.gcd*** requires two integer arguments and it also knows that expressions z and y are both of type int. therefore; it knows the call of gcd is legal. Furthermore it knows that gcd returns an int and therefore it knows that the assignment to x is legal.

Note Java has an important property: legal Java programs (that is, those accepted by the compiler) are guaranteed to be type safe. This mean that there cannot by any type errors when the program runs.

```
Object o1= a; // a is an array defined previously
```

```
Object o2= "abc";
```

Here **a** is an array and **s** is a string

An implication of the assignment rule is that *actual type* of the object obtained by the evaluating an expression is a subtype of the *apparent type* of the expression deduced by the compiler using declaration. For example, the *apparent type* of o2 is Object, but it actual type is *String.*

Type checking is always done using the apparent type. This means, for example, that any method calls made using the object will be determined to be legal based on the apparent type. Therefore only Object method like equal can be called on Object O2, string method like length (which returns a count of the number of characters in the string) cannot be called:

Example

```
If ( o2.equals("abc")// legal
If ( o2.lenght ( ) ) // illegal
```

Furthermore, the following is illegal:

```
String s=o2; // illegal
```

Because the apparent type of o2 is not a subtype of String. Compilation will fail when the program contains illegal code as in these examples.

Sometimes a program needs to determine the actual type of an object *at runtime*, for example, so that a method not provided by the apparent type can be called. This can be done by ***casting***.

```
s= (String) o2; // legal
```

The use of a cast causes to occur at runtime; if the check succeeds, the indicated computation is allowed and otherwise, a

ClassCastException will be raised. In the example, the casts check whether o2's actual type is the same as the indicated type string, these checks succeed, and therefore, the assignment of the statement is allowed.

Example:

```
Object o="abc";
```

We have a `following` code, we need to indicate whether or not a compile time error will occur, and for those statements that are legal at compile time, indicate whether they will return normally or by throwing an exception

```
boolean b= o.equals("a,b,c");
char c= o.charAt(1);
Object o2=b;
String s=o;
String t=(String)o;
          t.length()// 3
c=t.charAt(1);//
c=t.charAt(3);
```

The Object Class

The object class is the root of all classes in the Java technology programming language. If a class is declared with no extends clause, then the compiler adds implicitly the code extend Object to the declaration for example

```
Public class Employee {
                        // code here
                        }
Is equivalent to

Public class Employee extends Object {    // code goes here}
```

Note: `extends` is used for import other classes.

`Implements` is used import interface classes.

The `extends` *and* `implements` *clauses will discuss in GUI class.*

Two important methods are:

- equals
- toString

1. The equals Method

- The == operator determines if two ***references*** are identical to each other (that is, refer to the same object).
- The equals method determines if ***objects*** are *equal* but not necessarily identical.
- The Object implementation of the equals method uses the == operator

Example

```
Public class MyDate1 {
     private int day;
     private int month;
     private int year;
// constructor
public MyDate1(int day, int month, int year) {
     this.day = day; // this is used to assign
     values of instant variables
      this.month = month;
      this.year = year;

}
//@override
public boolean equals(Object o) {
    boolean result = false;
    if ((o != null) && (o instance of MyDate1))
           {
            MyDate1 d = (MyDate1) o;
     if ( (day == d.day) && (month == d.month)&&
     (year ==d.year) )
     {
```

```
          result = true;
              }// end if
       } // end if
       return result;
       } // end method

       }

public class TestEquals {
public static void main(String[] args) {

       MyDate1 date1 = new MyDate1(10, 10, 1976);
       MyDate1 date2 = new MyDate1(10, 10, 1976);
       if ( date1 == date2 ) {
       System.out.println("date1  is  identical  to
      date2");
       } else {
       System.out.println("date1 is not identical
       to date2");
        }
       if ( date1.equals(date2) ) {
       System.out.println("date1   is   equal   to
      date2");
        } else {
      System.out.println("date1  is  not  equal  to
      date2");

       System.out.println("set date2 = date1;");
       }
         date2 = date1;

       if ( date1 == date2 ) {
       System.out.println("date1 is identical to
       date2");
       } else {
       System.out.println("date1  is  not  identical
      to date2");
      }
  }
  }
This example generates the following output:
date1 is not identical to date2
date1 is equal to date2
```

```
set date2 = date1;
date1 is identical to date2
```

2. The toString method

It has the following characteristics:

- This method converts an object to a `String`.
- Use this method during string concatenation.
- Override this method to provide information about a user-defined object in readable format.

Q/ how we convert *primitive types* to a `String` using toSting method??

Convert Primitives Data Types to String

1.

In every Java program, you will encounter several primitive data types — byte, short, int, long, float, double, boolean or char— declared and manipulated. They drive the inner workings of most Java applications. **String** is a class in java that represents character strings.

```
int x = 5;  //  integer variable declaration
double d = 56.7; // double variable declaration
String s = "hello45678%";  // String object declaration
```

In Java, the primitive data types can be cast to a **String**. This means variable **x** and **d** in the above example can be turned to a String object and assigned to a String variable. This can be achieved in more than one ways.

- One way to cast a **long** primitive type to **String** is by using String concatenation. The '+' operator in java is overloaded as a String concatenator. Anything added to a String object using the '+' operator becomes a String.

```
class longToString1
{
Public static void main(String[] arg)
{
long  num1 = 4587;          // declare a long variable
String longString = " " + num1;  // concatenate the long and
an empty String
```

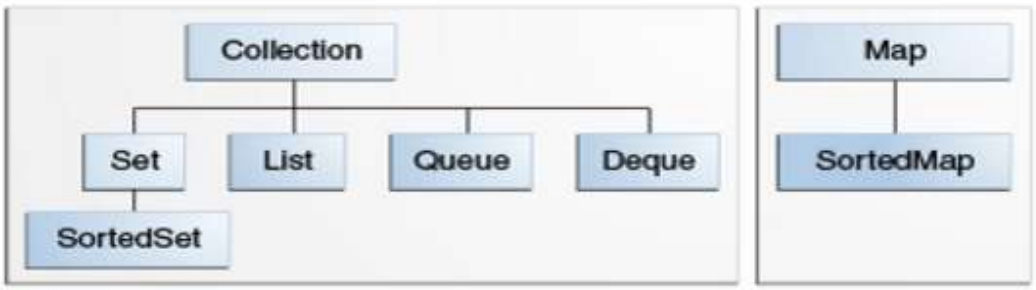
```
System.out.println(longString);
}
}
The output is a String with 4587
```

2.

- A second method to convert **long** to **String** is by using a static method of the class **Long**. The class **Long** is one of the *wrapper classes* in java that lets you wrap a primitive data type into an object. There is a matching wrapper class for every primitive type. Following are primitive data types and their corresponding wrapper classes:

Primitives Data Type	Wrapper Classes
• char • boolean • byte • int • double • short • long • float	Character Boolean Byte Integer Double Short Long Float

Wrapper classes are very useful when you want to add primitives to ***Collections***. You may want to add **long** variables to an arraylist. An arraylist, however, will not accept **long** variables. In such cases, you can wrap the **long** variable with a wrapper— turn a **long** variable to a **Long** object. The arraylist would now accept the long variables.



For example:

```
Wrapping a long variable:
long  x = 34533355;
Long  xWrap = new Long(x);
Unwrapping a Long object:
long xUnwrap = xWrap.longValue();
```

3.

- Wrapper classes have several static methods that will allow you to change a primitive variable to a String. A Java **long** type can be converted to **String** using the toString(long x).

```
class longToString2
{
Public static void main(String[] arg)
{
long  num2 = 4587;          // declare a long variable
String longString2 =  Long.toString(num2); // convert the long
variable num2 to a String
System.out.println(longString2);
}}
```

- We need a student to override the method toString () in his class to return different data types.

- Convert a String Str= “12345678” to long or others data types .