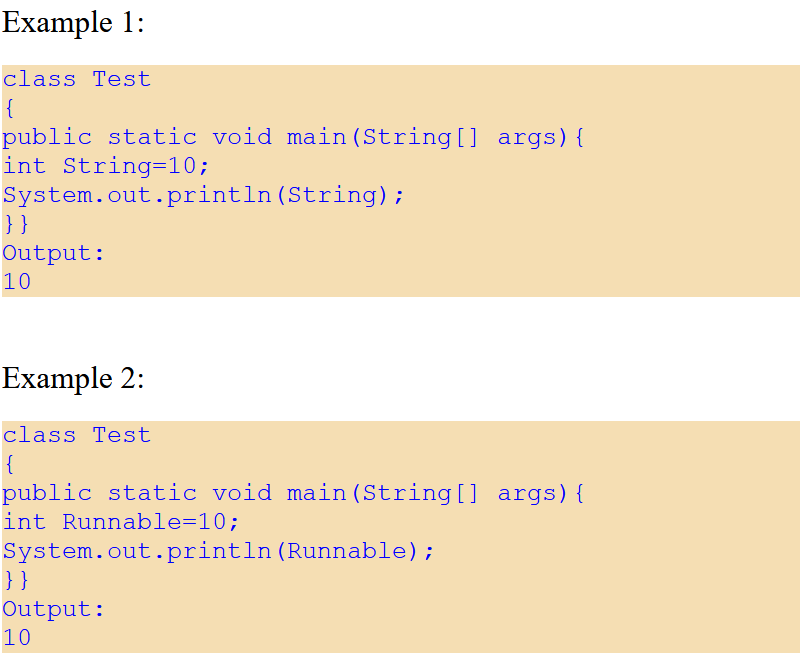
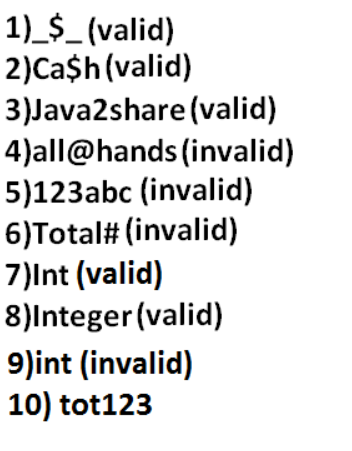
**Java Basics**

**Java Identifiers:**

A name in java program is called identifier. It may be class name, method name, variable name and label name

* a to z, A to Z, 0 to 9, \_ and $ can be used.
* Reserved words cannot be used.
* No length limit for identifiers.
* Class name can be used as identifiers, but it is not recommended.
* An identifiers cannot start with a number.

Java has 53 reserved words: 50 keywords and 3 reserved literals (true, false and null). Out of the 50 keywords 48 are used and 2 are unused (goto and const).

**Data Types:**

Java is a strongly typed programming language, which means every assignment is checked by the compiler for the type compatibility.

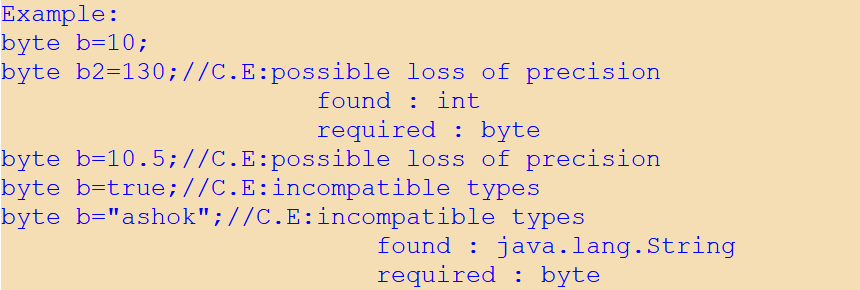
Java has 8 primitive data types: Numeric (Integral: byte, short, int, long; FloatingPoint: float, double), char and boolean.

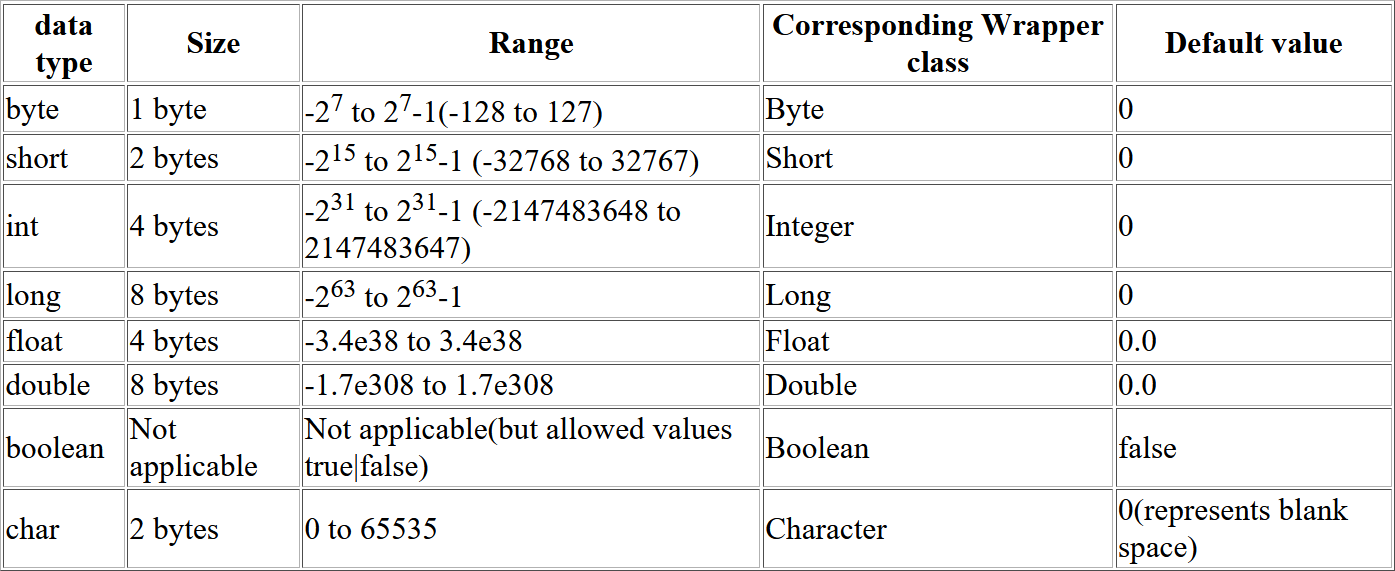
Except Boolean and char all remaining data types are considered as signed data types because we can represent both "+ve" and"-ve" numbers.

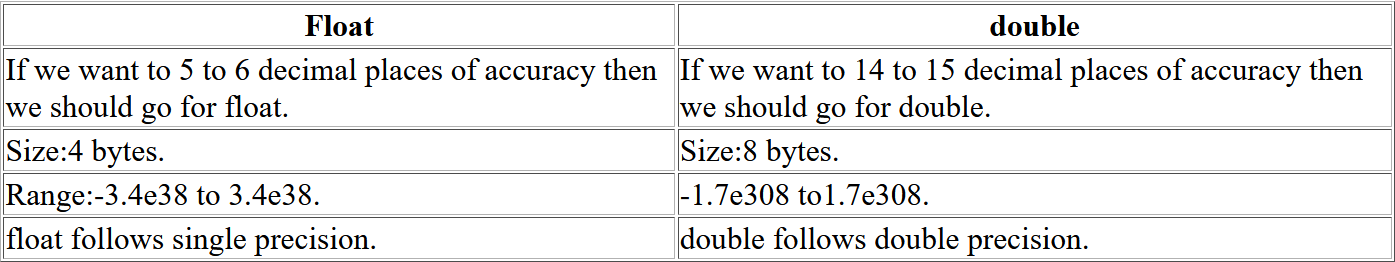
Short data type is best suitable for 16 bit processors like 8086 but these processors are completely outdated.

The return type of length() method is long.

In old languages like C & C++ are ASCII code based the no.Of ASCII code characters are **< 256** to represent these 256 characters 8 - bits enough hence char size in old languages 1 byte.   
  
In java we are allowed to use any worldwide alphabets character and java is Unicode based and no. of unicode characters are **> 256 and <= 65536** to represent all these characters one byte is not enough compulsory we should go for 2 bytes.







**Literals:**

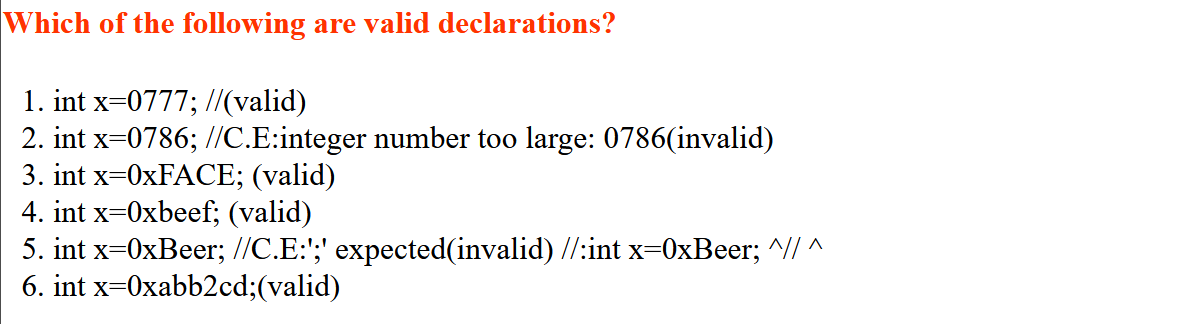
Any constant value which can be assigned to the variable is called literal.

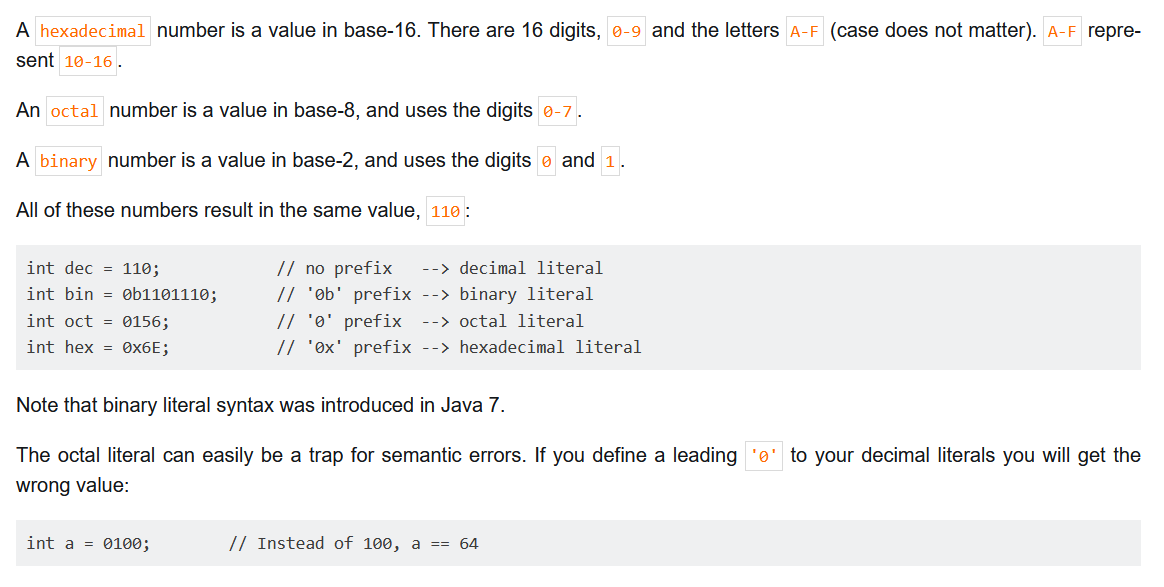
**Integral Literals:** For the integral data types (byte, short, int and long) we can specify literal value in the following ways.

1) Decimal literals: Allowed digits are 0 to 9.   
Example: int x=10;   
  
2) Octal literals: Allowed digits are 0 to 7. Literal value should be prefixed with zero.  
Example: int x=010;   
  
3) Hexa Decimal literals:

* The allowed digits are 0 to 9, A to Z.
* For the extra digits we can use both upper case and lowercase characters.
* This is one of very few areas where java is not case sensitive.
* Literal value should be prefixed with 0x(or)0X.

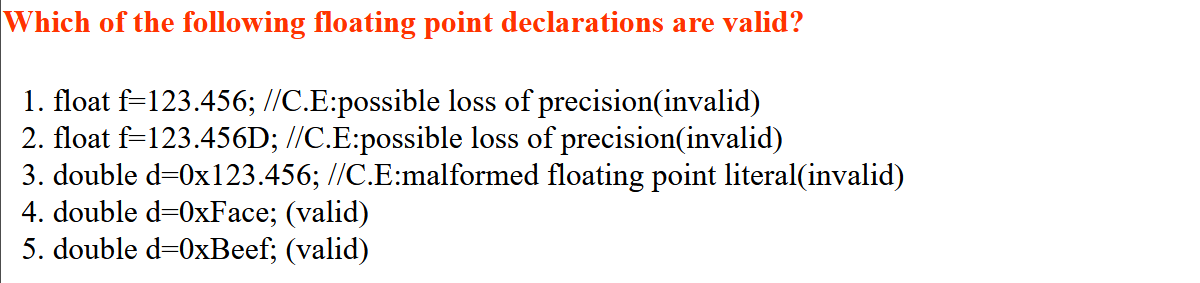
Example: int x=0x10;   
These are the only possible ways to specify integral literal.



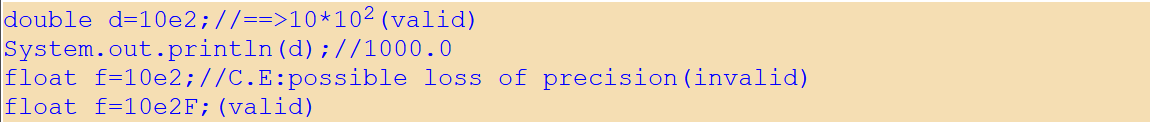


**Floating Point Literals:** Floating point literal is by default double type but we can specify explicitly as float type by suffixing with f or F. We can specify explicitly floating point literal as double type by suffixing with d or D.

We can specify floating point literal only in decimal form and we can't specify in octal and hexadecimal forms. But we can assign integral literal directly to the floating point data types and that integral literal can be specified in decimal , octal and Hexa decimal form also.

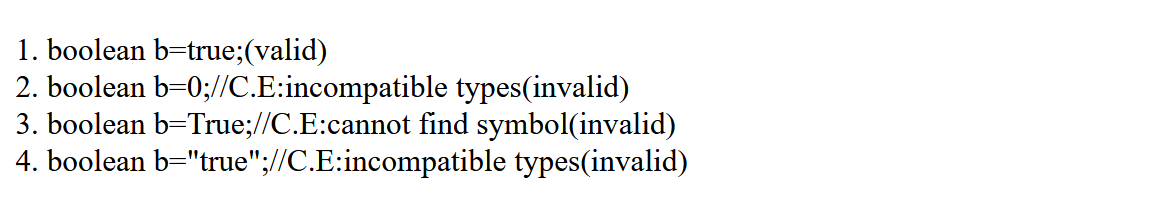


We can specify floating point literal even in exponential form also(significant notation).



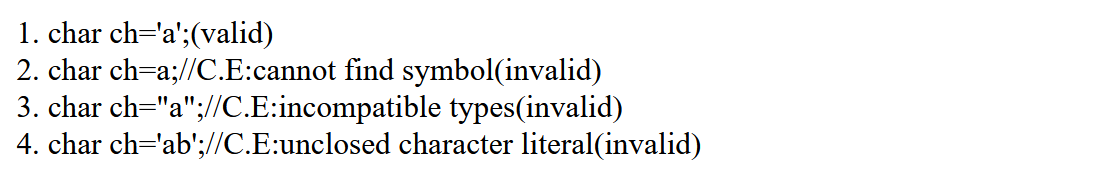
**Boolean Literals:**

The only allowed values for the boolean type are true (or) false where case is important. i.e., lower case.

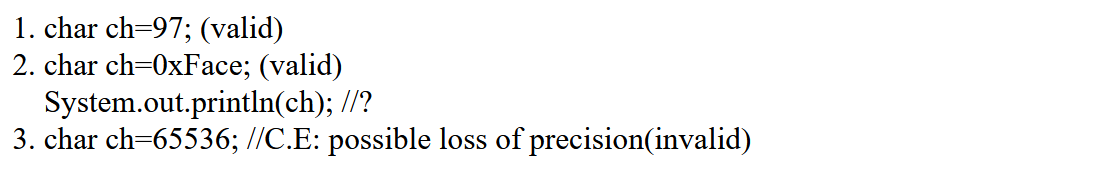


**Char Literals:**

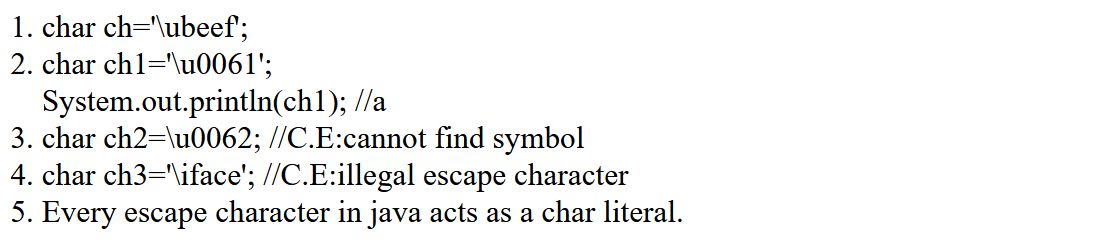
A char literal can be represented as single character within single quotes.



We can specify a char literal as integral literal which represents Unicode of that character.  
We can specify that integral literal either in decimal or octal or hexadecimal form but allowed values range is 0 to 65535.



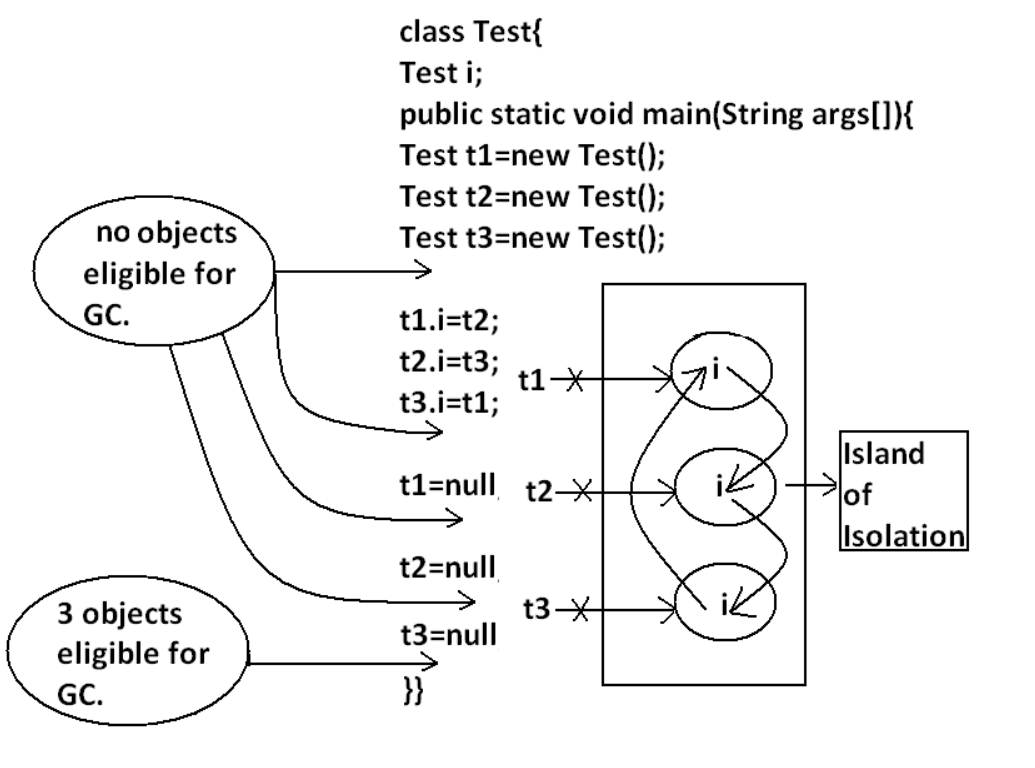
We can represent a char literal by Unicode representation which is nothing but **‘\uxxxx'** (4 digit hexa-decimal number).



**Garbage Collection**

An object is eligible for GC if and only if it does not have any references.

* Assigning "null" to all its reference variable.
* Reassign all its reference variables to some other object.
* Objects created inside a method are by default eligible for GC once method completes.
* Island of Isolation strategy (all references are internal references) says that even though object having reference still it is eligible for GC. Example:

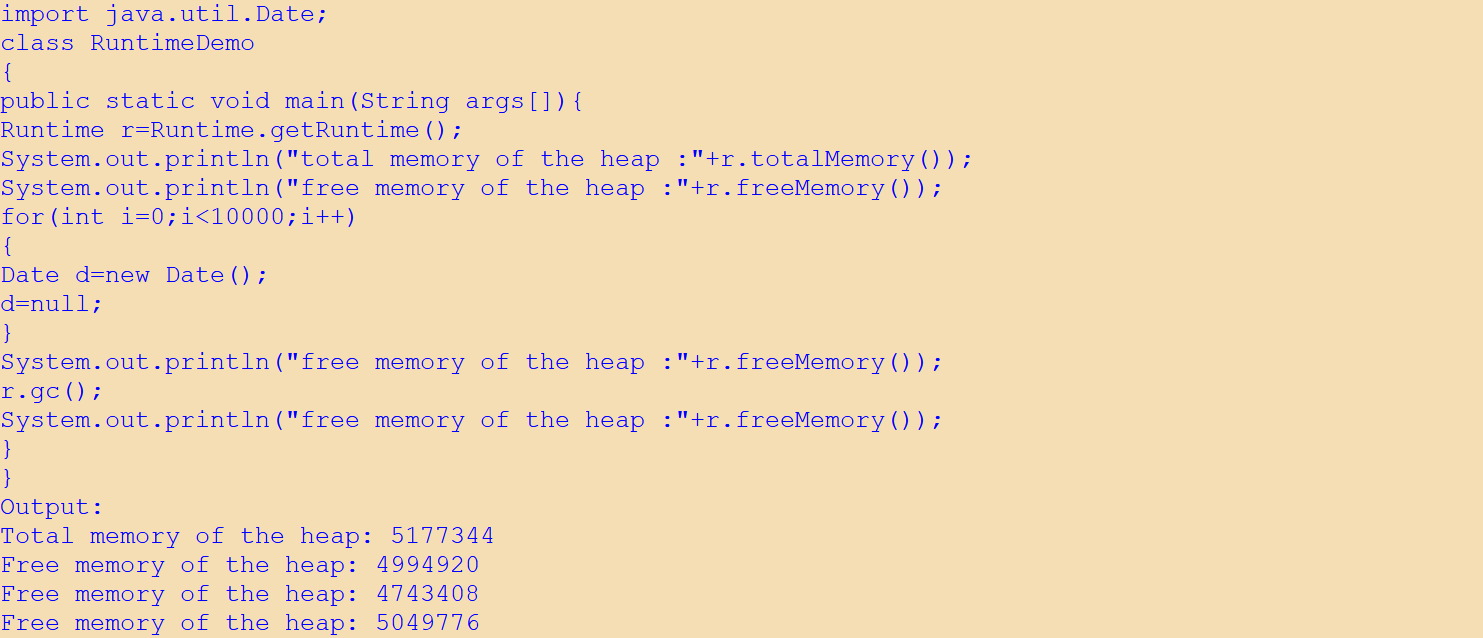


The following are various ways to request JVM to run GC explicitly:

1. By System class’s static method: System.gc();
2. Using an object of Runtime class. It is a singleton class present in java.lang package.

Example: **Runtime r=Runtime.getRuntime();**

**freeMemory():** returns the free memory present in the heap.  
**totalMemory():** returns total memory of the heap.  
**gc():** for requesting jvm to run gc.



**Which of the following are valid ways for requesting JVM to run GC?**System.gc(); (valid)   
Runtime.gc(); (invalid)   
(new Runtime).gc(); (invalid)   
Runtime.getRuntime().gc(); (valid)

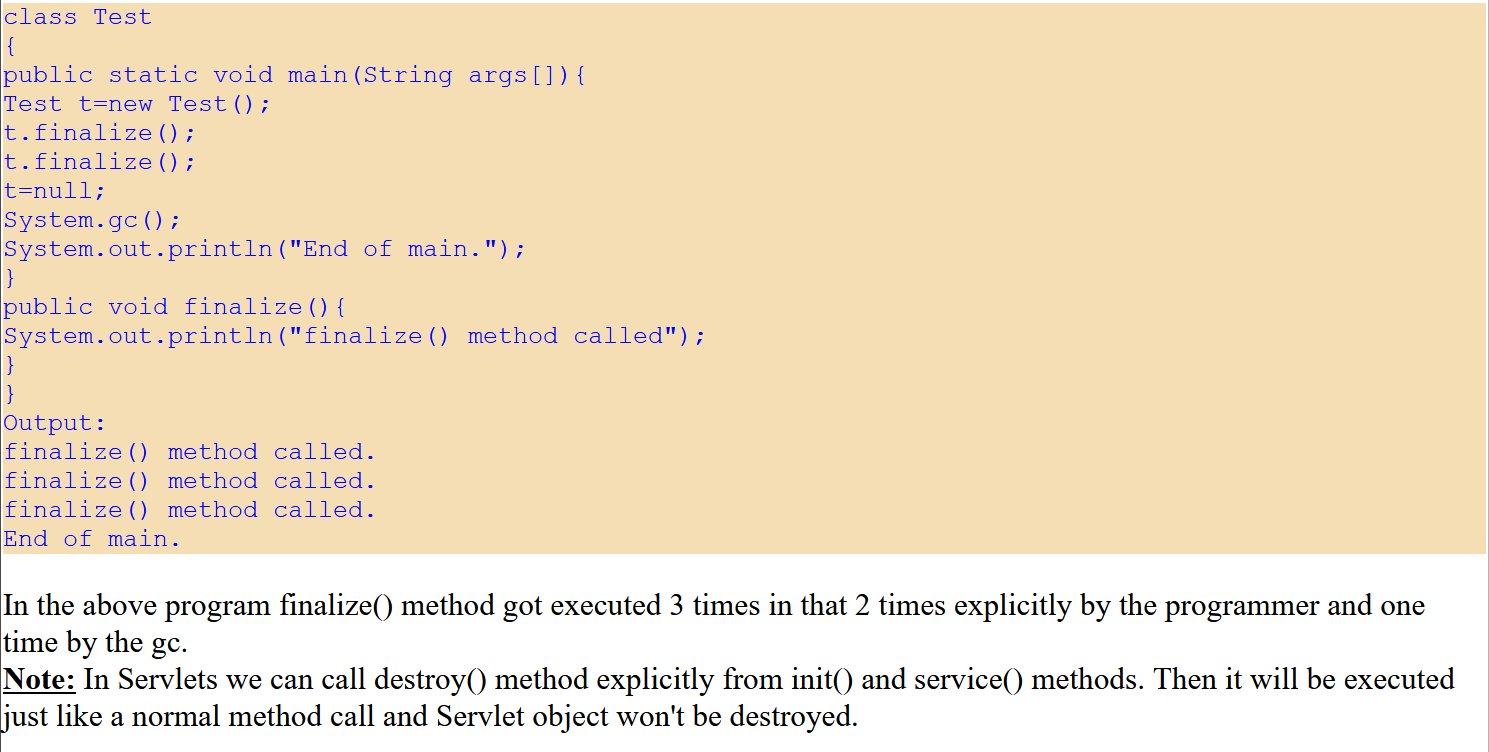
**Note:** Over Runtime class gc() method , System class gc() method is recommended to use.

**Note:** In java it is not possible to find size of an object and address of an object.

**Finalization:** Just before destroying any object gc always calls finalize() method to perform cleanup activities. If the corresponding class contains finalize() method then it will be executed otherwise Object class finalize() method will be executed.

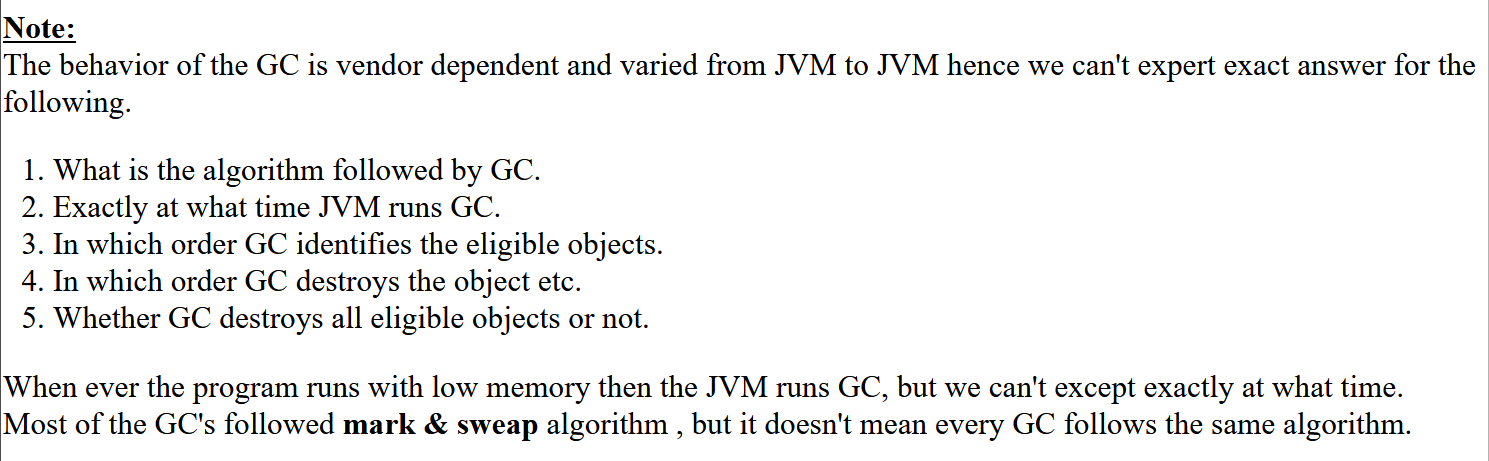
Method signature: protected void finalize() throws Throwable

We can call finalize() method explicitly, then it will be executed just like a normal method call.



If the programmer calls explicitly finalize() method and while executing the finalize() method if an exception raised and uncaught then the program will be terminated abnormally.  
If GC calls finalize() method and while executing the finalize()method if an exception raised and uncaught then JVM simply ignores that exception and the program will be terminated normally.

**Which of the following is true?**   
While executing finalize() method JVM ignores every exception(invalid).   
While executing finalize() method JVM ignores only uncaught exception(valid).



**Memory Leaks:**

A Memory Leak is a situation **where there are objects present in the heap that are no longer used, but the garbage collector is unable to remove them from memory because they are still referenced,** and therefore, they're unnecessarily maintained.

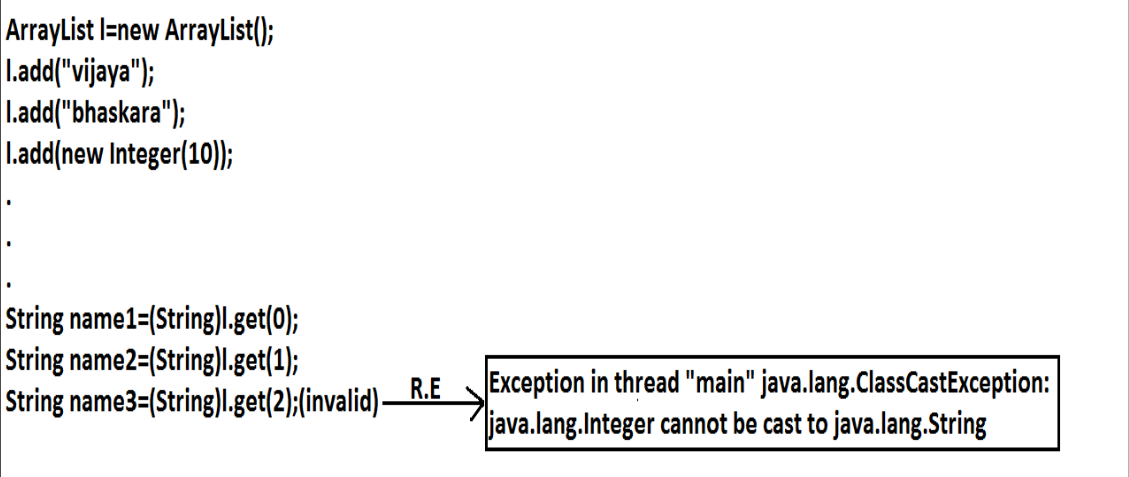
* Memory leaks through static fields: In Java, static **fields have a life that usually matches the entire lifetime of the running application** (unless ClassLoader becomes eligible for garbage collection).

**Generics**

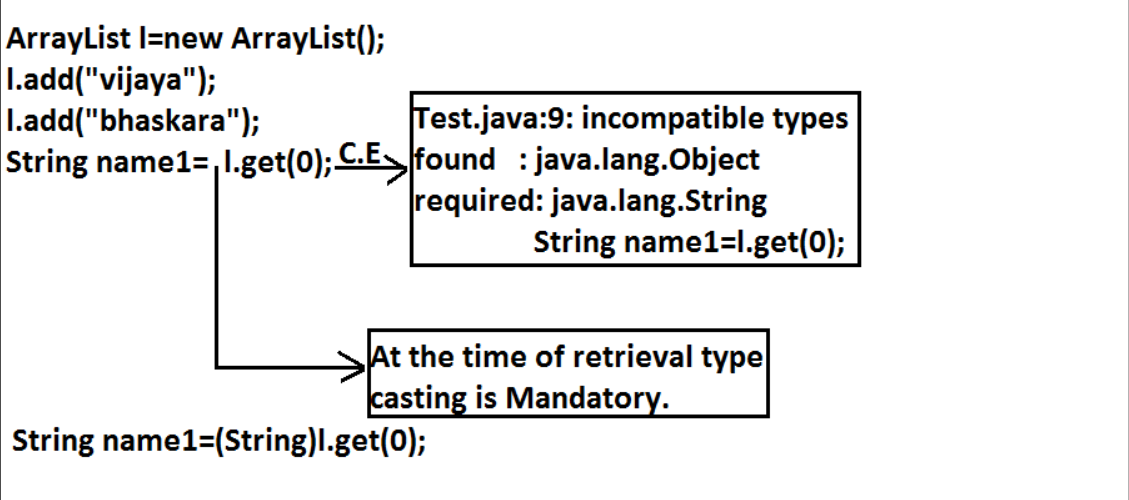
The main objective of Generics is to provide **Type-Safety** and to resolve **Type-Casting** problems.

Type-Safety Issue: Arrays are always type safe that is we can give the guarantee for the type of elements present inside array. But collections are not type safe that is we can't provide any guarantee for the type of elements present inside collection.

Example: No compile time error, but we get runtime exception.



Type-casting Issue: In the case of array at the time of retrieval it is not required to perform any type casting. But in the case of collection at the time of retrieval compulsory we should perform type casting otherwise we will get compile time error.



Conclusions:

* Polymorphism concept is applicable only for the base type but not for parameter type.
* for the parameter type we can use any class or interface name but not primitive value(type). Otherwise, we will get compile time error.

Generic classes: Class with a type parameter. Example: class Account<T>

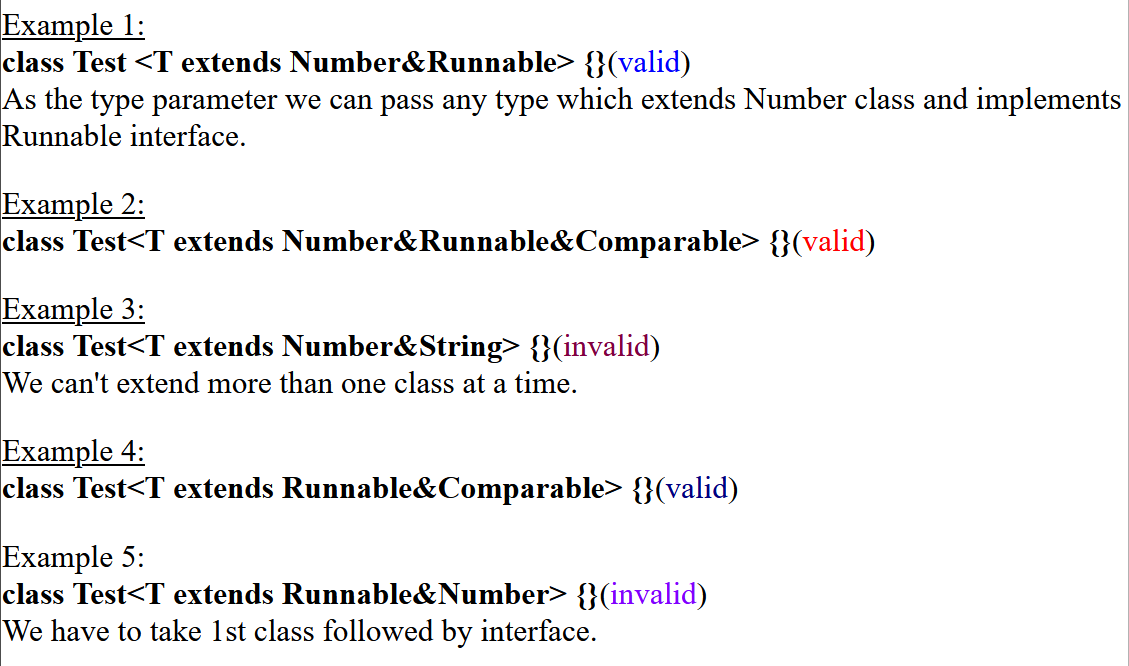
Bounded generic types for Class: We can bound the type parameter for a particular range by using extends keyword.

Examples:

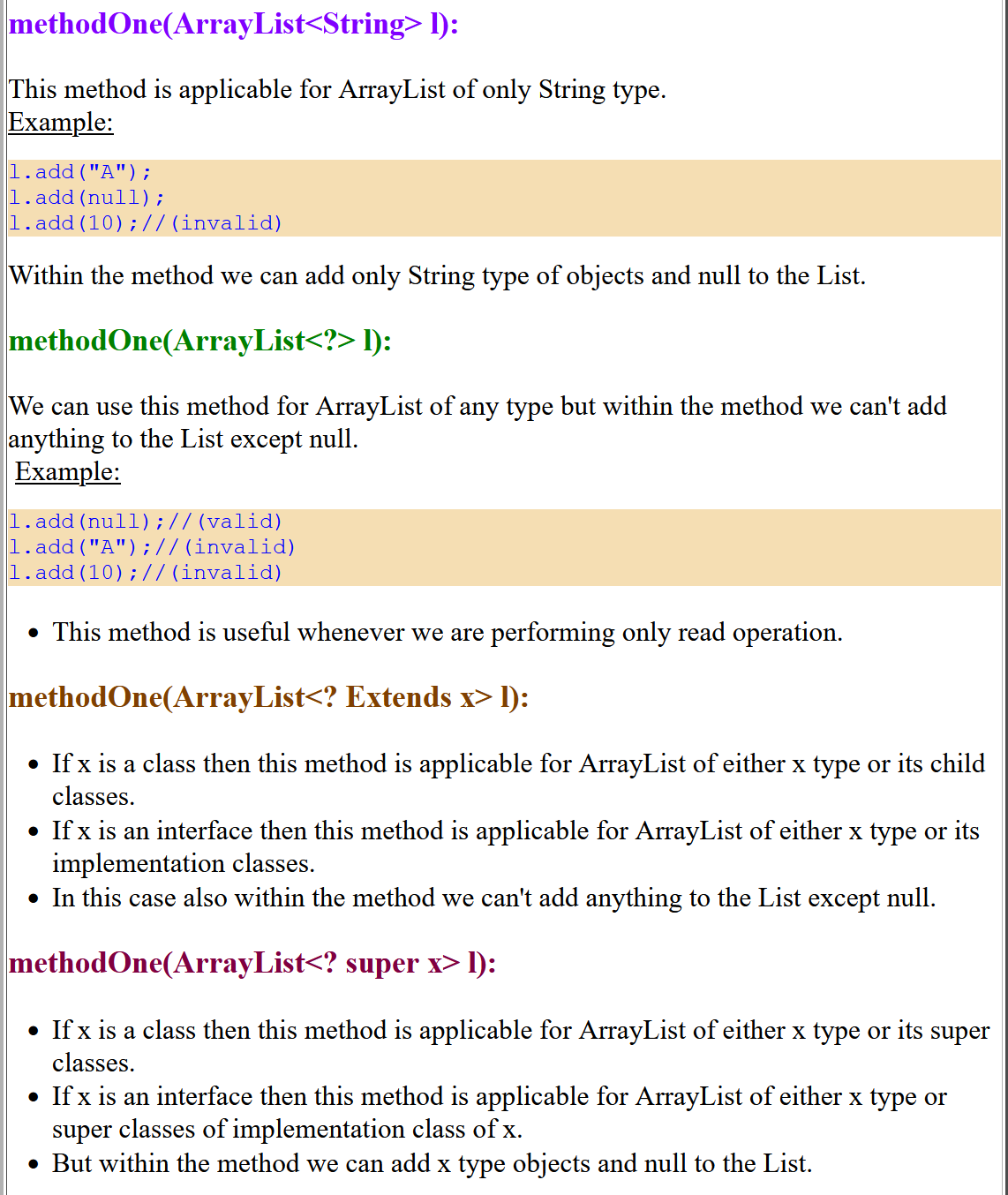
1. class Test<T extends X>

If x is a class, then as the type parameter, we can pass either x or its child classes. If x is an interface, then as the type parameter, we can pass either x or its implementation classes.

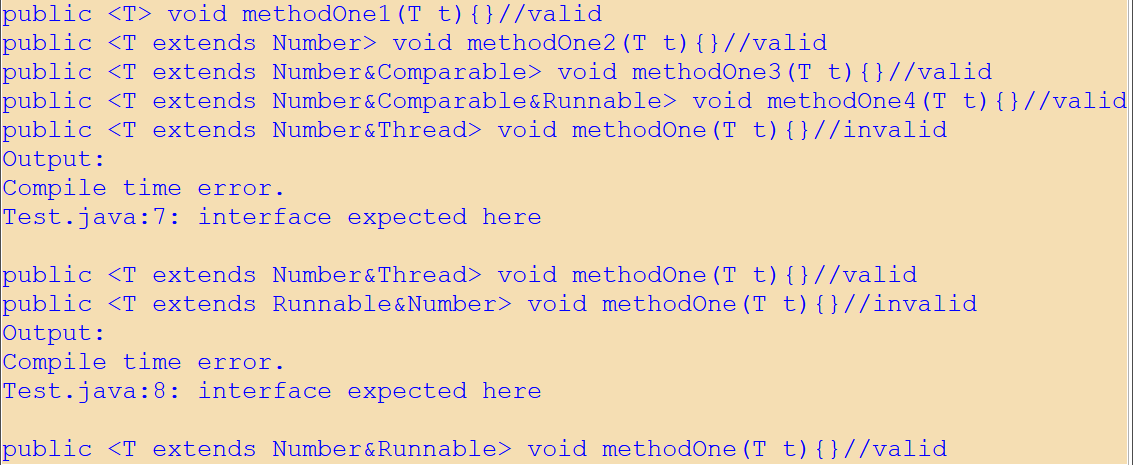
We can pass any no of type parameters; it need not be one. Example: **HashMap<Integer,String> h=new HashMap<Integer,String>();**

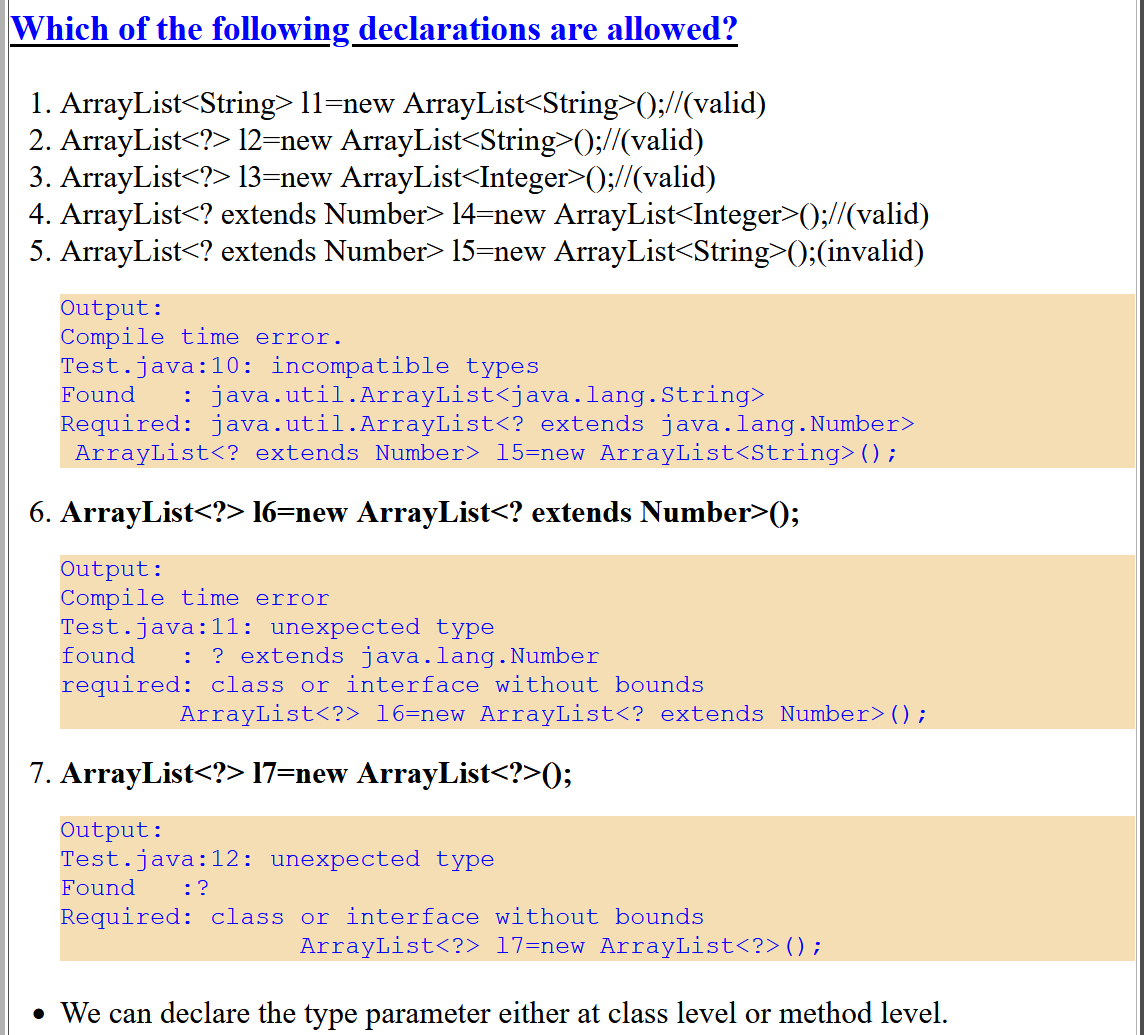


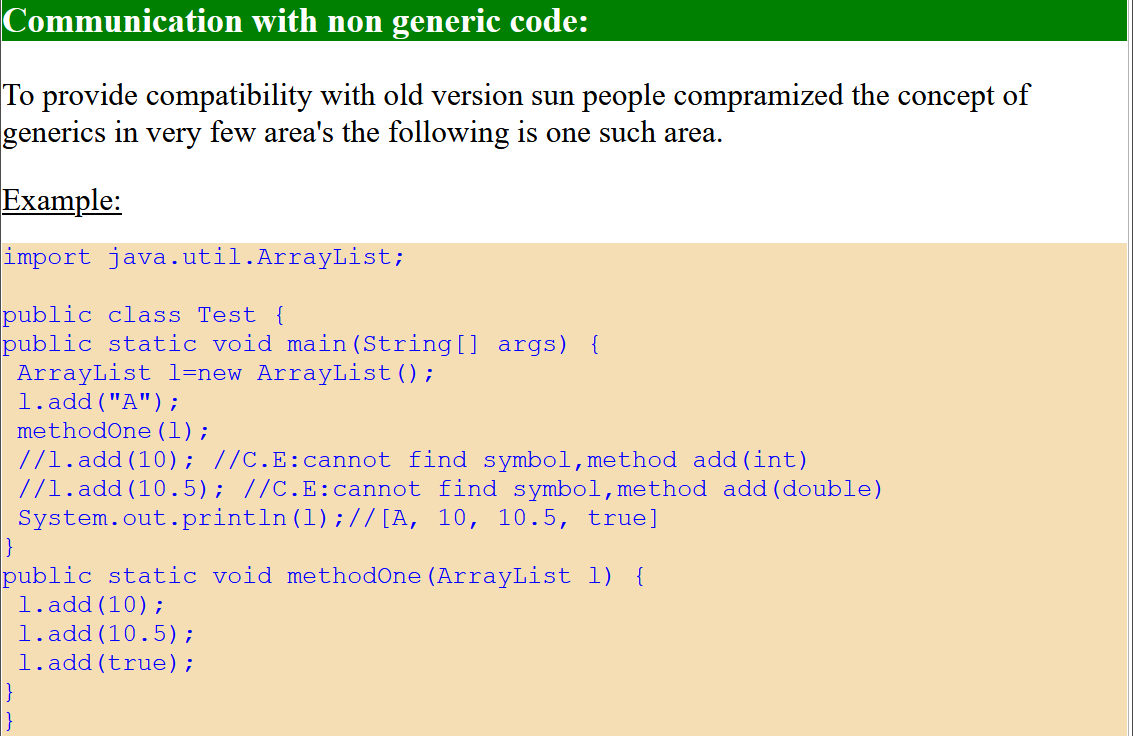
Bounded generic types for Methods: Uses wildcard character (?) along with extends/supers keyword.



We have to declare generic type just before return type of a method.

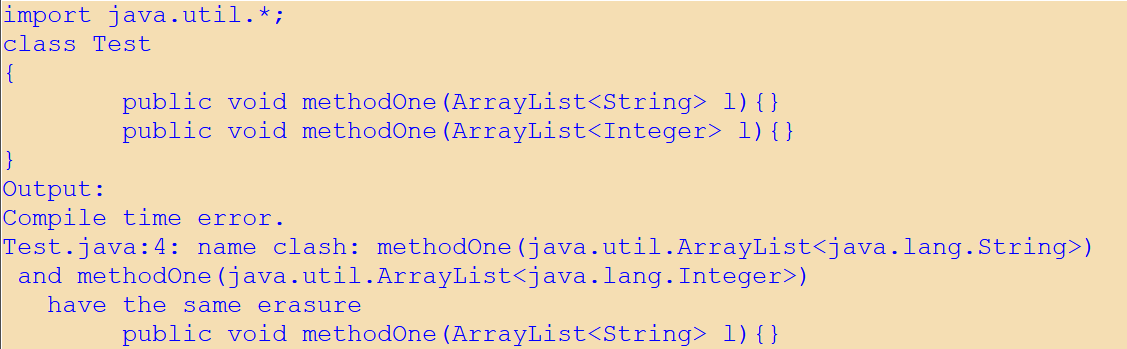




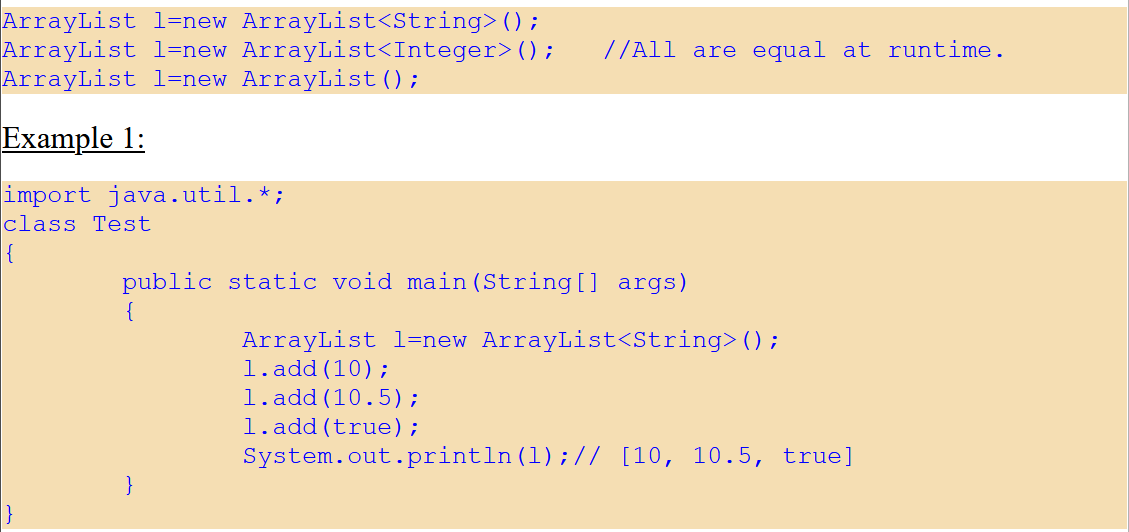


Runtime of Generics: Generics concept is applicable only at compile time, at runtime there is no such type of concept.

Proof:



The following declarations are equal, as the compiler resolves



Similarly, the following two declarations are equal:

