Core Java Foundation – Day03

Akshita Chanchlani

Inheritance

- If "is-a" relationship is exist between the types then we should use inheritance.
- Inheritance is also called as generalization.
- Example
    1. Manager is a employee
    2. Book is a product
    3. Triangle is a shape
    4. SavingAccount is a account.

```
class Employee{ //Parent class
    //TODO
}
class Manager extends Employee{ //Child class
    //TODO
}
//Here class Manager is extended from class Employee.
```

Inheritance

- If we want to implement inheritance then we should use extends keyword.
- In Java, parent class is called as super class and child class is called as sub class.
- Java do not support private and protected mode of inheritance
- If Java, class can extend only one class. In other words, multiple class inheritance is not allowed.
- Consider following code:

```
class A{    }
class B{    }
class C extends A, B{    //Not OK
}
```

Inheritance

- During inheritance, if super type and sub type is class, then it is called as implementation inheritance.

Single implementation Inheritance

```
class A{    }
class B extends A{ }    //OK
```

Hierarchical implementation Inheritance

```
class A{    }
class B extends A{ } //OK
class C extends A{ } //OK
```

Multiple implementation Inheritance

```
class A{    }
class B{    }
class C extends A, B{ } //Not OK
```

Multilevel implementation inheritance

```
class A{    }
class B extends A{ } //OK
class C extends B{ } //OK
```

Multiple Inheritance In Java

```
class A{    }
class B{    }
class C extends A, B{   //Not OK : Multiple implementation inheritance
    //TODO
}
```

```
interface A{    }
interface B{    }
interface C extends A, B{   //OK : Multiple interface inheritance
    //TODO
}
```

```
interface A{    }
interface B{    }
class C implements A, B{   //OK : Multiple interface implementation inheritance
    //TODO
}
```

Inheritance

- If we create instance of sub class then all the non static fields declared in super class and sub class get space inside it. In other words, non static fields of super class inherit into sub class.
- Static field do not get space inside instance. It is designed to share among all the instances of same class.
- Using sub class, we can access static fields declared in super class. In other words, static fields of super class inherit into sub class.
- All the fields of super class inherit into sub class but only non static fields gets space inside instance of sub class.
- Fields of sub class, do not inherit into super class. Hence if we create instance of super class then only non static fields declared in super class get space inside it.
- If we declare field in super class static then, all the instances of super class and sub class share single copy of static field declared in super class.

Inheritance

- We can call/invoke, non static method of super class on instance of sub class. In other words, non static method inherit into sub class.

- We can call static method of super class on sub class. In other words, static method inherit into sub class.

- Except constructor, all the methods of super class inherit into sub class.

Inheritance

- If we create instance of super class then only super class constructor gets called. But if we create instance of sub class then JVM first give call to the super class constructor and then sub class constructor.
- From any constructor of sub class, by default, super class's parameterless constructor gets called.
- Using super statement, we can call any constructor of super class from constructor of sub class.
- Super statement, must be first statement inside constructor body.

Inheritance

- According to client's requirement, if implementation of super class is logically incomplete / partially complete then we should extend the class. In other words we should use inheritance.
- According to client's requirement, if implementation of super class method is logically incomplete / partially complete then we should redefine method inside sub class.
- Process of redefining method of super class inside sub class is called as method overriding.

Inheritance

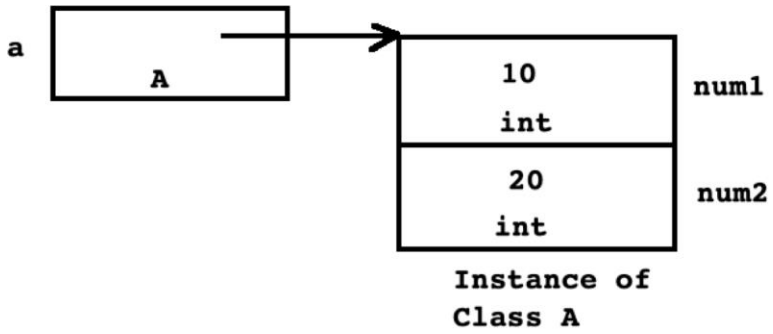
- If name of super class and sub class method is same and if we try to call such method on instance of sub class then preference is given to the method of sub class. This is called shadowing.
- Using super keyword, we can access members of super class inside method of sub class.
- During inheritance, members of sub class do not inherit into super class. Hence using super class instance, we can access members of super class only.
- During inheritance, members of super class, inherit into sub class. Hence using sub class instance, we can access members of super class as well as sub class.

Inheritance

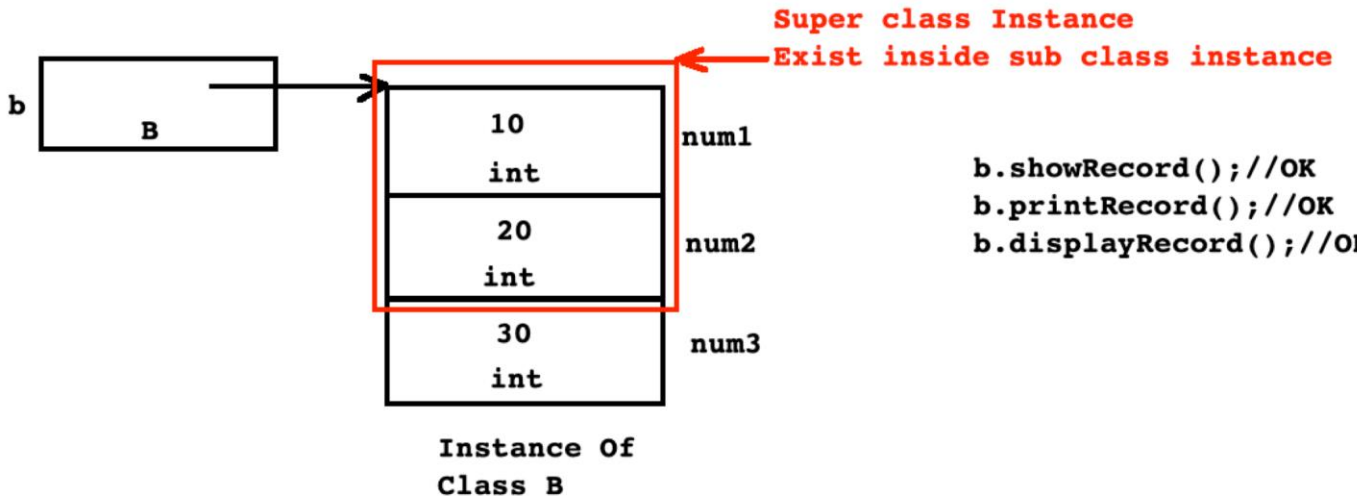
- Members of super class, inherit into sub class. Hence we can consider, sub class instance as a super class instance.
- Example : Employee is a Person
  - Since Employee contains all the properties and behavior of person hence Employee is a person.
- Since sub class instance can be considered as super class instance, we can use it in place super class instance.

```
Employee emp1 = null;    //OK

Employee emp2 = new Employee( );    //OK

person p = new Person( );
Employee emp3 = p;    //NOT OK

Employee emp4 = new Person( );  //NOT OK
```

Inheritance




```
a.showRecord();//OK
a.printRecord();//OK
a.displayRecord();//NOT OK
```




```
b.showRecord();//OK
b.printRecord();//OK
b.displayRecord();//OK
```

Upcasting

- Process of converting reference of sub class into reference of super class is called as upcasting.

```
Employee emp = null;
Person p = ( Person )emp;    //OK : Upcasting
//p : null
//emp : null
```

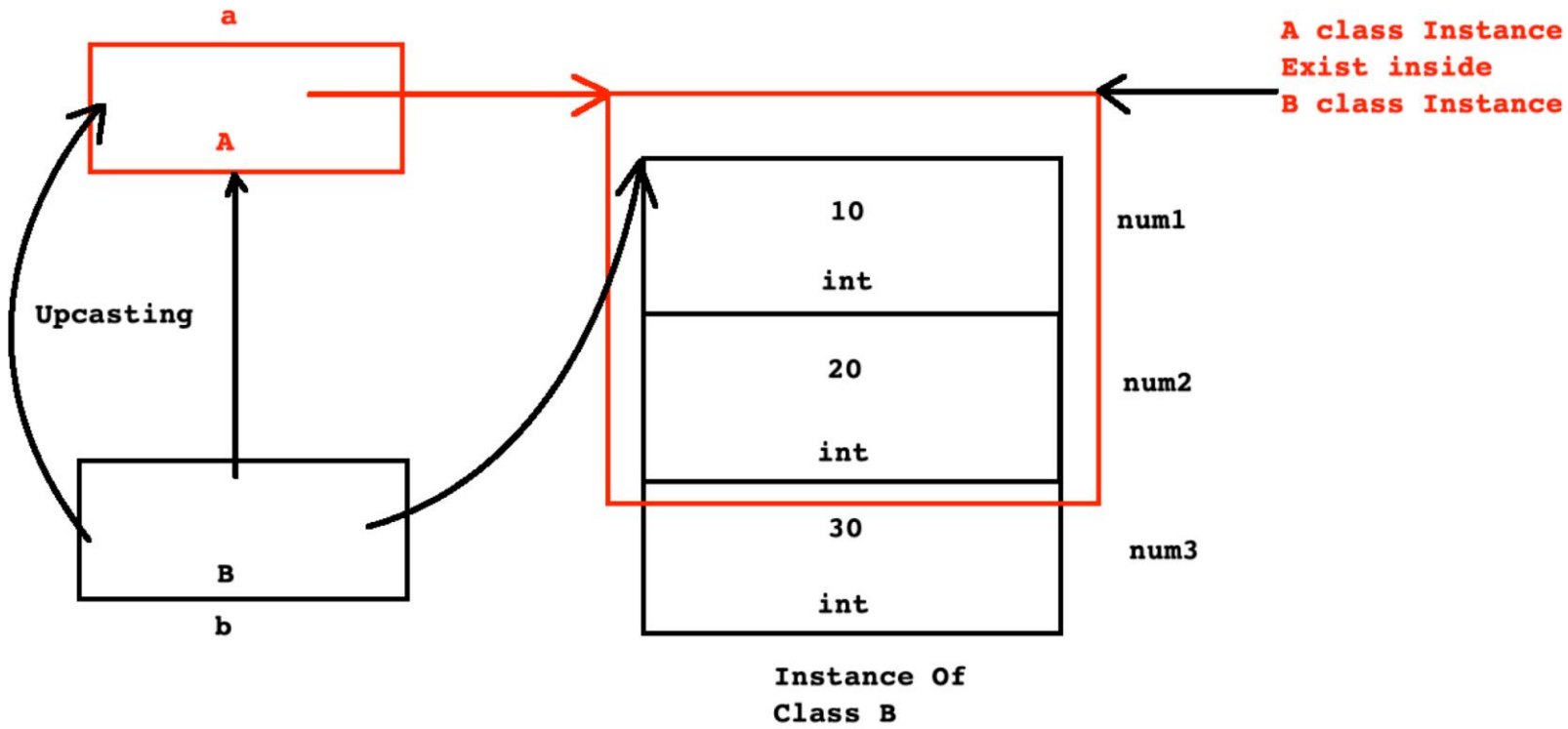
```
Employee emp = new Employee( );
//Person p = ( Person )emp;    //OK : Upcasting
Person p = emp;    //OK : Upcasting
```

- Using upcasting, we can minimize object dependency in the code. In other words, we can reduce maintenance of the code.

- During upcasting, explicit type casting is optional.

- Super class reference variable can contain reference of sub class instance. It is also called as upcasting.
  - **Person p = new Employee( ); //Upcasting**

Upcasting




```
B b = new B( );
A a = ( A )b; //Upcasting

B b = new B( );
A a = b;  //OK : Upcasting
```

```
A a = new B( ); //OK : Upcasting
```

Upcasting

- In case of upcasting, using super class reference variable, we can access:
  1. Fields of super class
  2. Methods of super class
  3. overridden methods of sub class

- In case of upcasting, using super class reference variable, we can not access:
  1. fields of sub class
  2. Non overridden methods of sub class.

- If we want to access fields and non overridden methods of sub class then we should do down casting.

Down Casting

- Process of converting reference of super class into reference of sub class is called down casting.

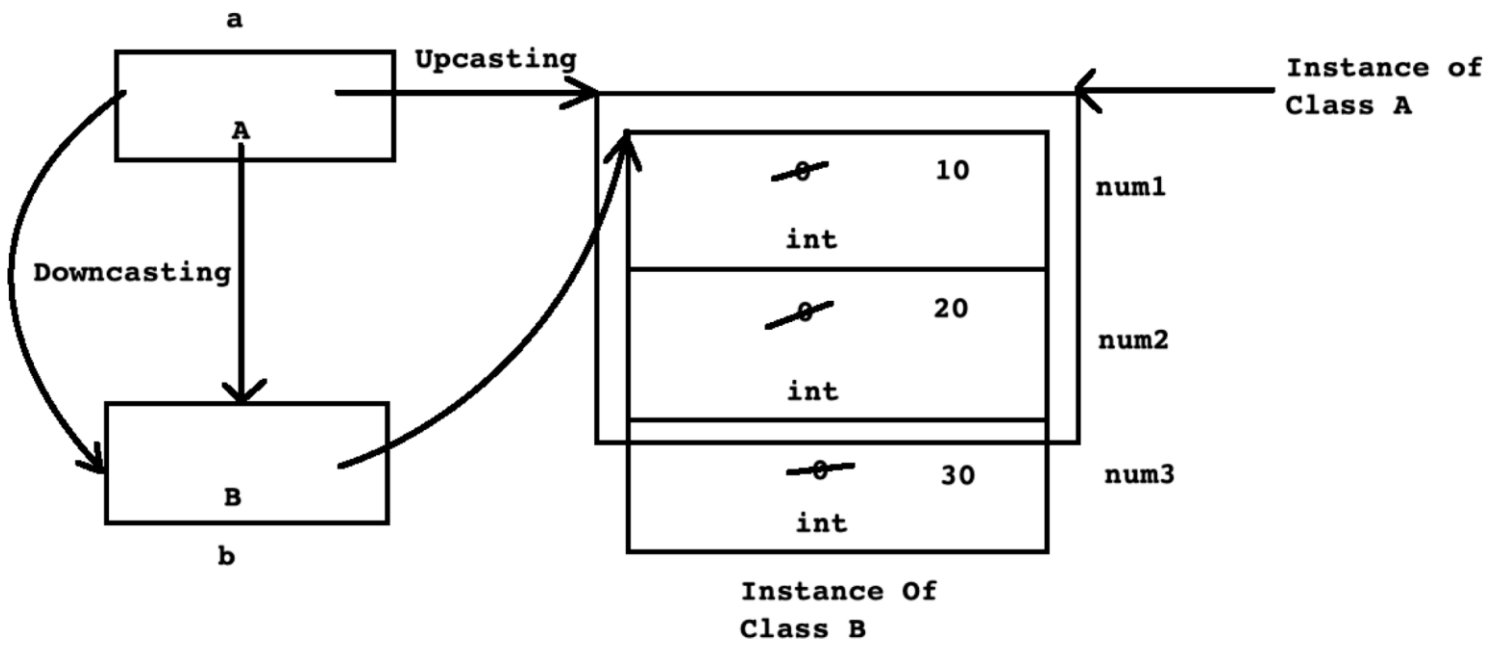
```
Person p = new Employee( ); //Upcasting
Employee emp = ( Employee)p;    //Downcasting : OK
```

```
Person p = null;
Employee emp = ( Employee)p; //OK : Downcasting
//p = null
//emp = null
```

```
Person p = new Person();
Employee emp = ( Employee)p; //Downcasting : ClassCastException
```

- Only in case of upcasting, we should do down casting. Otherwise JVM will throw ClassCastException.
- In case of upcasting, using super class reference variable, we can access overridden method of sub class. It is also called as dynamic method dispatch.

Down Casting



Method Overriding

- Process of redefining method of super class inside sub class is called method overriding.
- Method redefined in sub class is called overrided method.
- When we call method of sub class using reference of super class then it is called dynamic method dispatch.

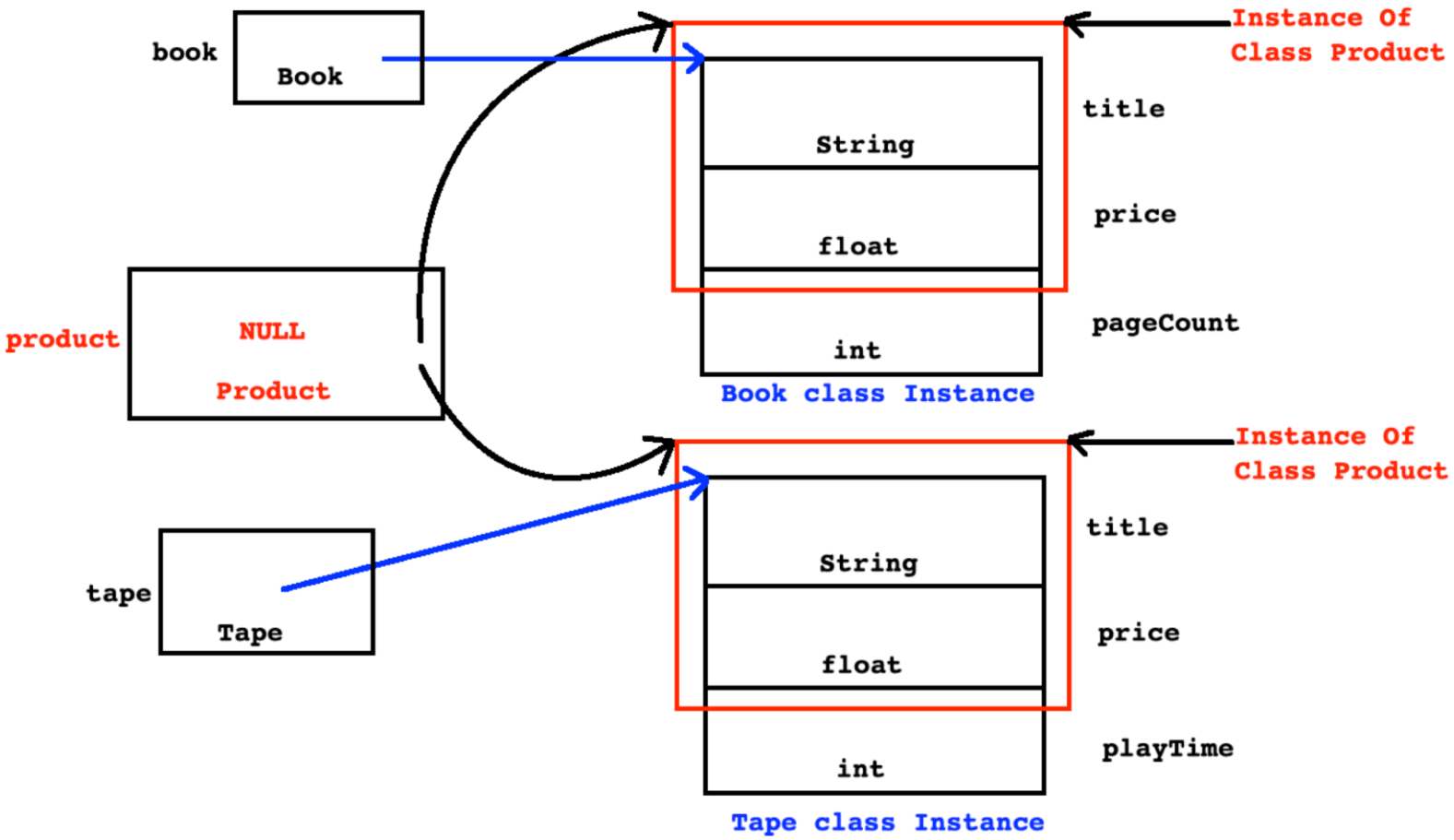
```
class A{
    public void print( ) {
        System.out.println("A.print");
    }
}
class B extends A{
    public void print( ) {
        System.out.println("B.print");
    }
}
public class Program {
    public static void main(String[] args) {
        A a = new B( ); //Upcasting
        a.print();  //Dynamic Method Dispatch
    }
}
```

Method Overriding

- If implementation of super class method is partially complete then we should redefine/override method inside sub class.
- **Rules of method overriding**
  1. Access modifier in sub class method should be same or it should be wider.
  2. Return type of sub class method should be same or it should be sub type. In other words, it should be covariant.
  3. Name of the method, number of parameters and type of parameters inside sub class method must be same.
  4. Checked exception list inside sub class method should be same or it should be sub set.

- Override is annotation declared in java.lang package.
- We can use this annotation on method only.
- It helps developer to override method inside sub class.

Down Casting



equals( ) Method

- equals is non final method of java.lang.Object class.
- Syntax:
  - **public boolean equals(Object obj);**
- Consider its implementation inside Object class.

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

- If we do not define equals method inside class then super class's equals method gets called.
- equals method of Object class do not compare state of instances. It compares state of references.
- If we want to compare state of instances then we should override equals method inside class.

java.lang.Character

- It is a final class declared in java.lang package.
- The Character class wraps a value of the primitive type char in an object.
- This class provides a large number of static methods for determining a character's category (lowercase letter, digit, etc.) and for converting characters from uppercase to lowercase and vice versa.
- The fields and methods of class Character are defined in terms of character information from the Unicode Standard.
- The char data type are based on the original Unicode specification, which defined characters as fixed-width 16-bit entities.

java.lang.Character

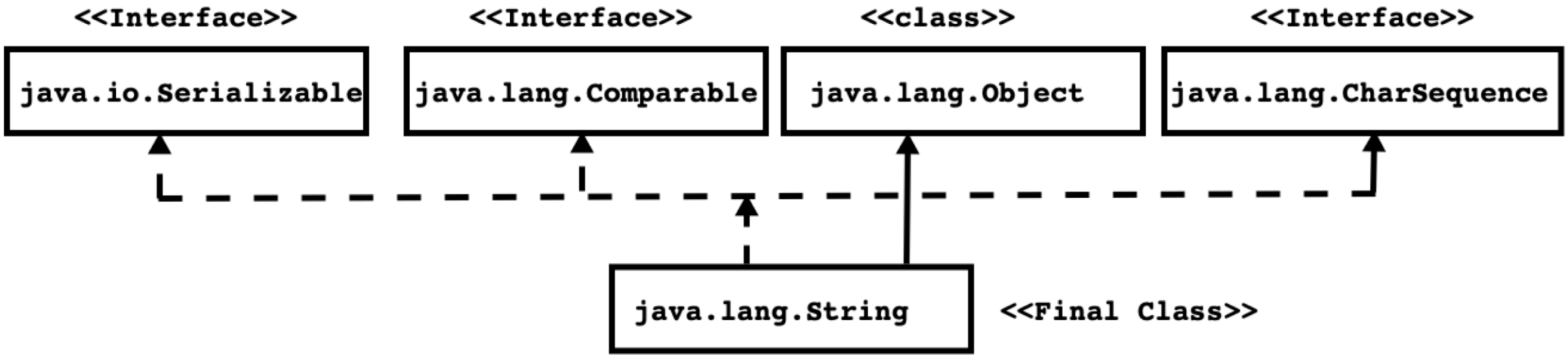
- The range of legal *code points* is now U+0000 to U+10FFFF, known as *Unicode scalar value*.
- The set of characters from U+0000 to U+FFFF is sometimes referred to as the *Basic Multilingual Plane (BMP)*.
- Characters whose code points are greater than U+FFFF are called *supplementary characters*.
- The Java platform uses the UTF-16 representation in char arrays and in the String and StringBuffer classes.
- A char value, therefore, represents Basic Multilingual Plane (BMP) code point
- https://medium.com/jspoint/introduction-to-character-encoding-3b9735f265a6

String Introduction

- Strings, which are widely used in Java programming, are a sequence of characters.

- In the Java programming language, strings are objects.
- We can use following classes to manipulate string:
    1. java.lang.String
    2. java.lang.StringBuffer
    3. java.lang.StringBuilder
    4. java.util.StringTokenizer
    5. java.util.regex.Pattern
    6. java.util.regex.Matcher

String Class Hierarchy



String Introduction

- Serializable is a Marker interface declared in java.io package.
- Comparable is Functional interface declared in java.lang package.
  1. int compareTo(T other)
- CharSequence is interface declared in java.lang package.
  1. char charAt(int index)
  2. int length()
  3. CharSequence subSequence(int start, int end)
  4. default IntStream chars()
  5. default IntStream codePoints( )
- Object is non final, concrete class declared in java.lang package.
  1. It is having 11 methods(5 Non final + 6 final)
- String is a final class declared in java.lang package.

String Introduction

- String is not a built-in or primitive type. It is a class, hence considered as non primitive/reference type.

- We can create instance of String with and W/o new operator.
  - String str = new String("Sandeep"); //String Instance
  - String str = "SunBeam"; //String Literal

- String str = "SunBeam", is equivalent to:
- ```
  char[] data = new char[ ]{'S','u','n','B','e','a','m'};
  ```
- ```
  String str = new String( data );
  ```

String concatenation

- If we want to concatenate Strings then we should use concat() method:
  - "public String concat(String str)"

- Consider following Example:

```
String s1 = "SunBeam";
String s2  =  "Pune/Karad";
String s3 = s1.concat( s2 );
```

- The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

```
String s1 = "SunBeam";
String s2 = s1 +" Pune/Karad";
```

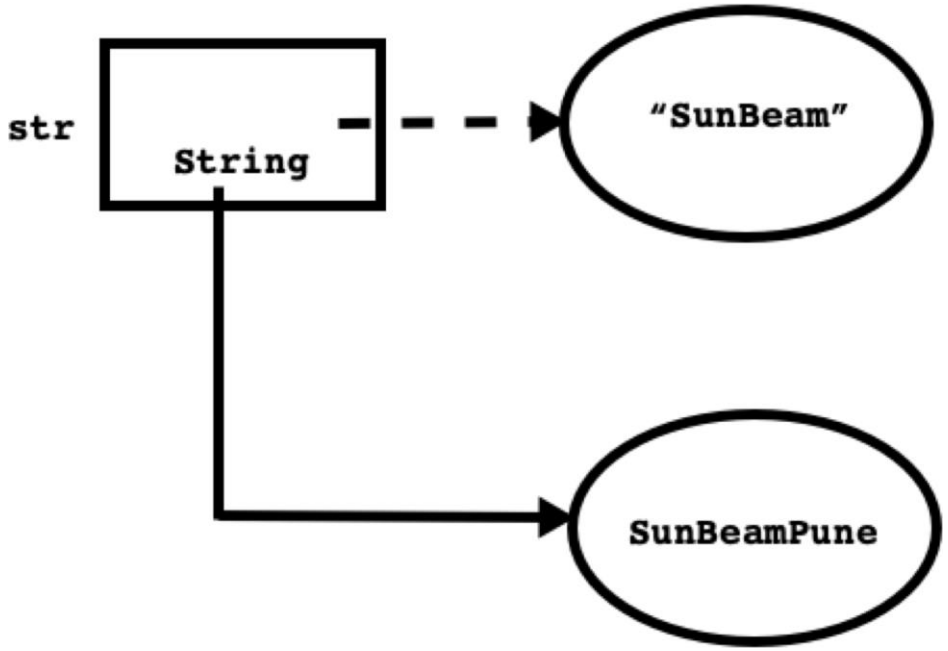
-

- `int pinCode = 411057;`
- `String str = "Pune,"+pinCode;`

Immutable Strings

- Strings are constant; their values cannot be changed after they are created.
- Because String objects are immutable they can be shared.

```
String str = "SunBeam";

str = str + "Pune";
```



A Strategy for Defining Immutable Objects

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
   - Don't provide methods that modify the mutable objects.
   - Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

String Class Constructors

1. `public String()`

2. `public String(byte[] bytes)`

3. `public String(char[] value)`

4. `public String(String original)`

5. `public String(StringBuffer buffer)`

6. `public String(StringBuilder builder)`

String Class Methods

1. public char charAt(int index)
2. public int compareTo(String anotherString)
3. public String concat(String str)
4. public boolean equalsIgnoreCase(String anotherString)
5. public boolean startsWith(String prefix)
6. public boolean endsWith(String suffix)
7. public static String format(String format, Object... args)
8. public byte[] getBytes()
9. public int indexOf(int ch)
10. public int indexOf(String str)
11. public String intern( )
12. public boolean isEmpty()
13. public int length()
14. public boolean matches(String regex)

String Class Methods

15.public String[] split(String regex)

16.public String substring(int beginIndex)

17.public String substring(int beginIndex, int endIndex)

18.public char[] toCharArray()

19.public String toLowerCase()

20.public String toUpperCase()

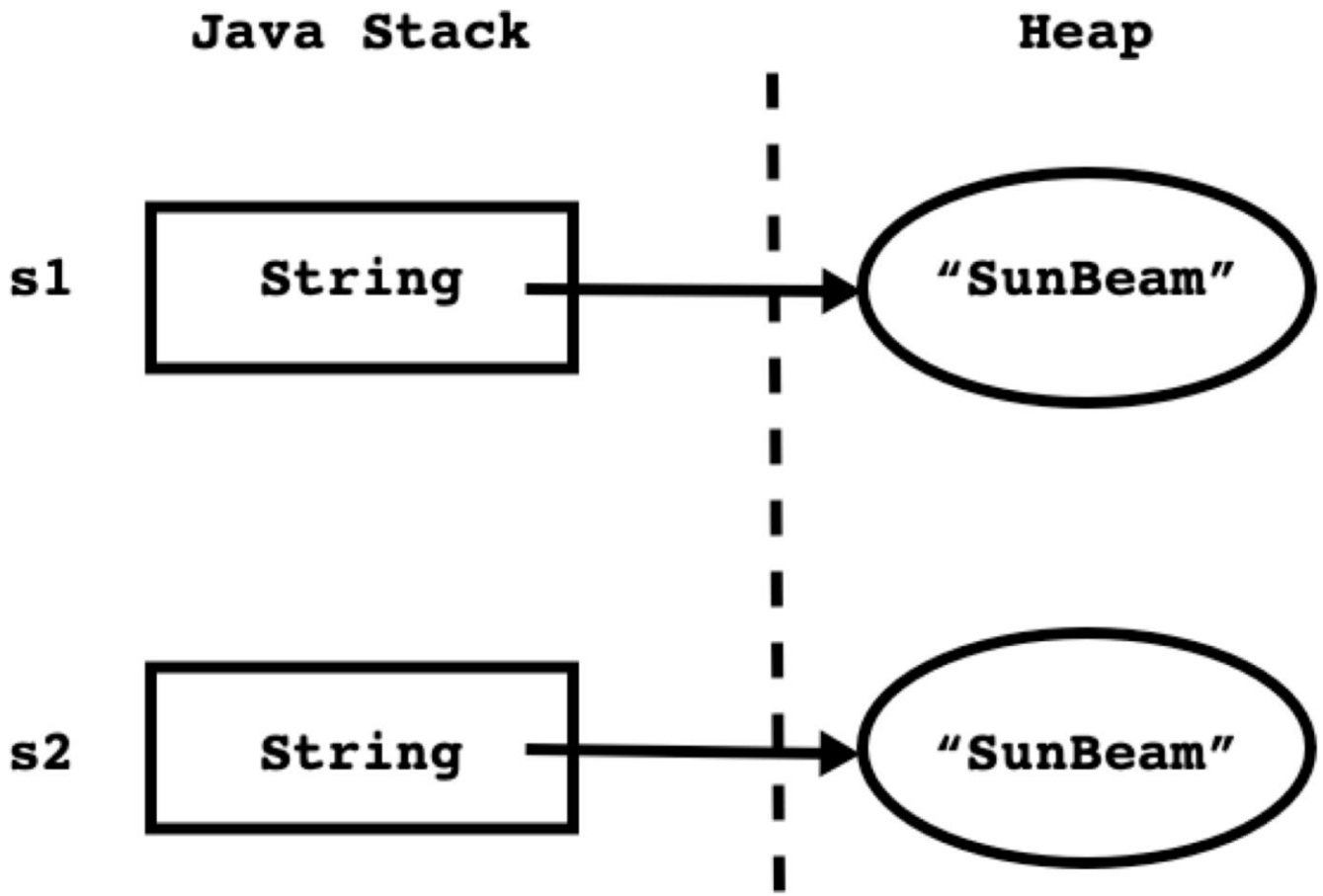
21.public String trim()

22.public static String valueOf(Object obj)

String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = new String("SunBeam");
      String s2 = new String("SunBeam");
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

String Twister



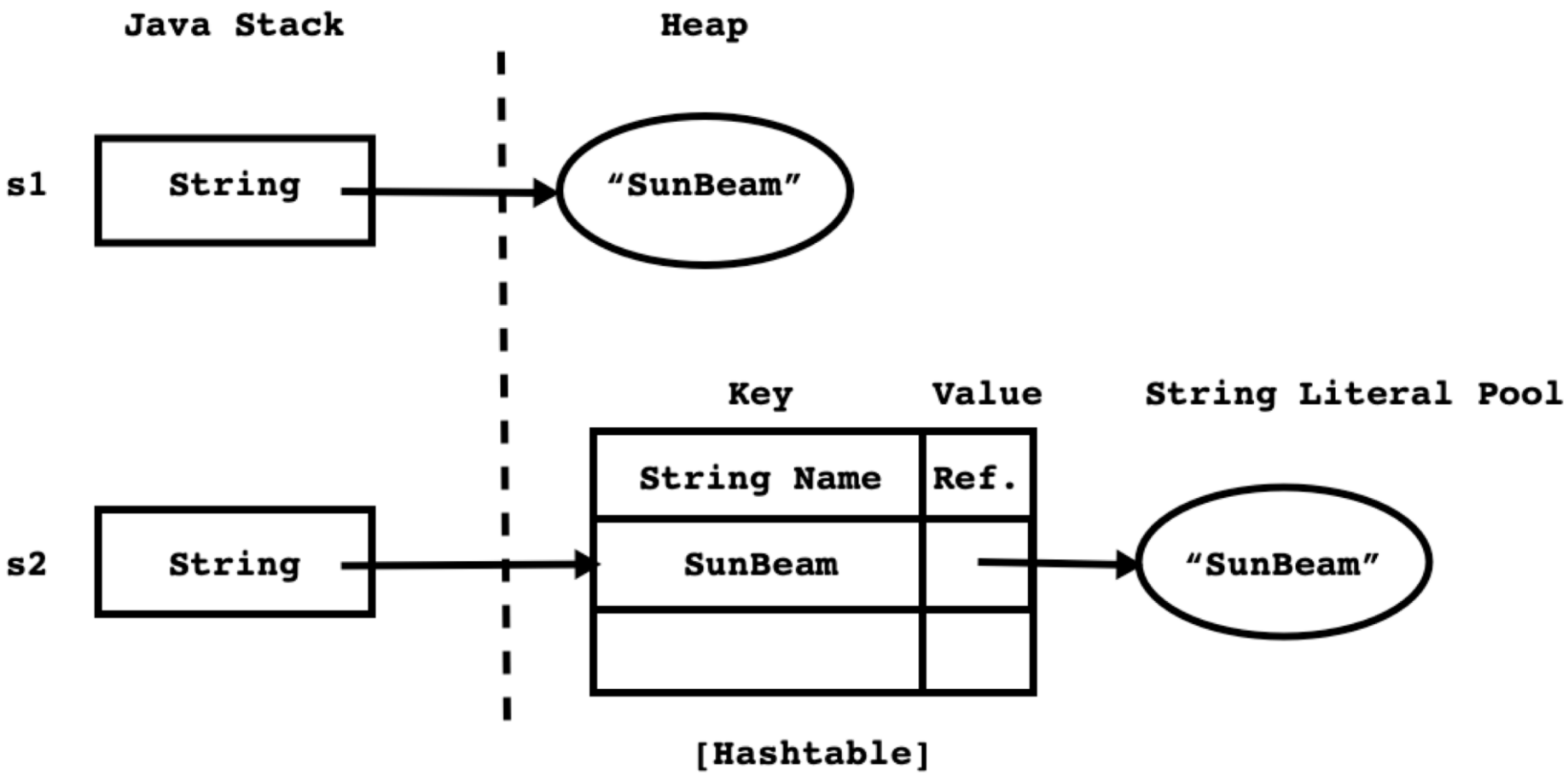
String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = new String("SunBeam");
      String s2 = new String("SunBeam");
      if( s1.equals(s2) )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Equal
   }
}
```

String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = new String("SunBeam");
      String s2 = "SunBeam";
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

String Twister



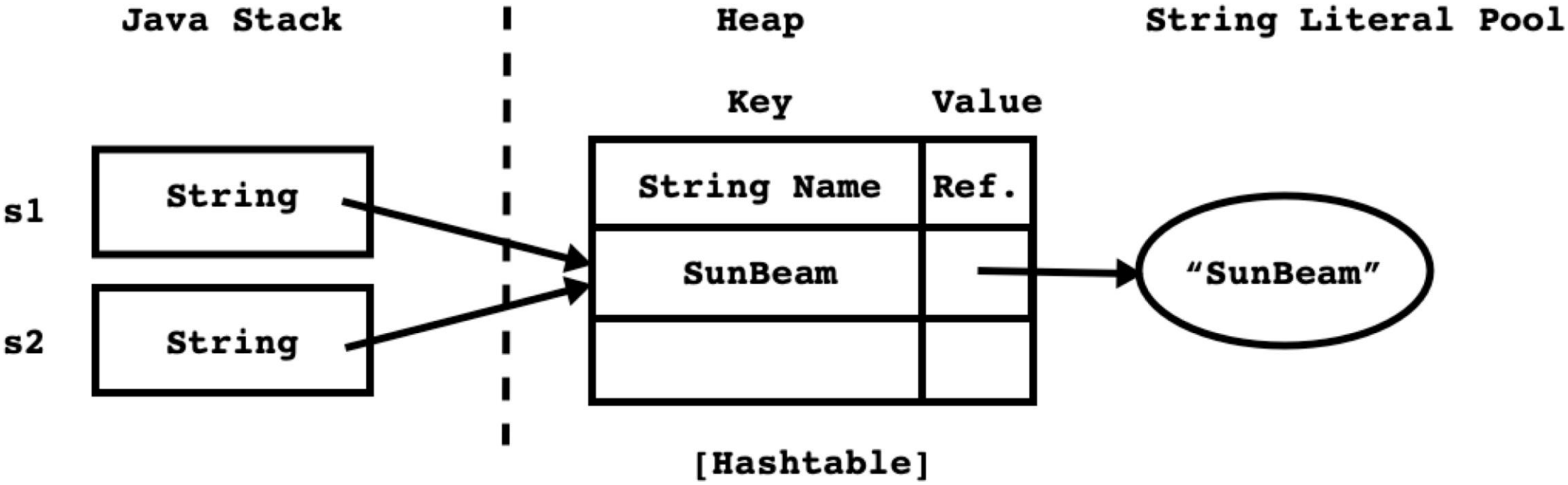
String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = new String("SunBeam");
      String s2 = "SunBeam";
      if( s1.equals(s2) )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Equal
   }
}
```

String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = "SunBeam";
      String s2 = "SunBeam";
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Equal
   }
}
```

String Twister



String Twister

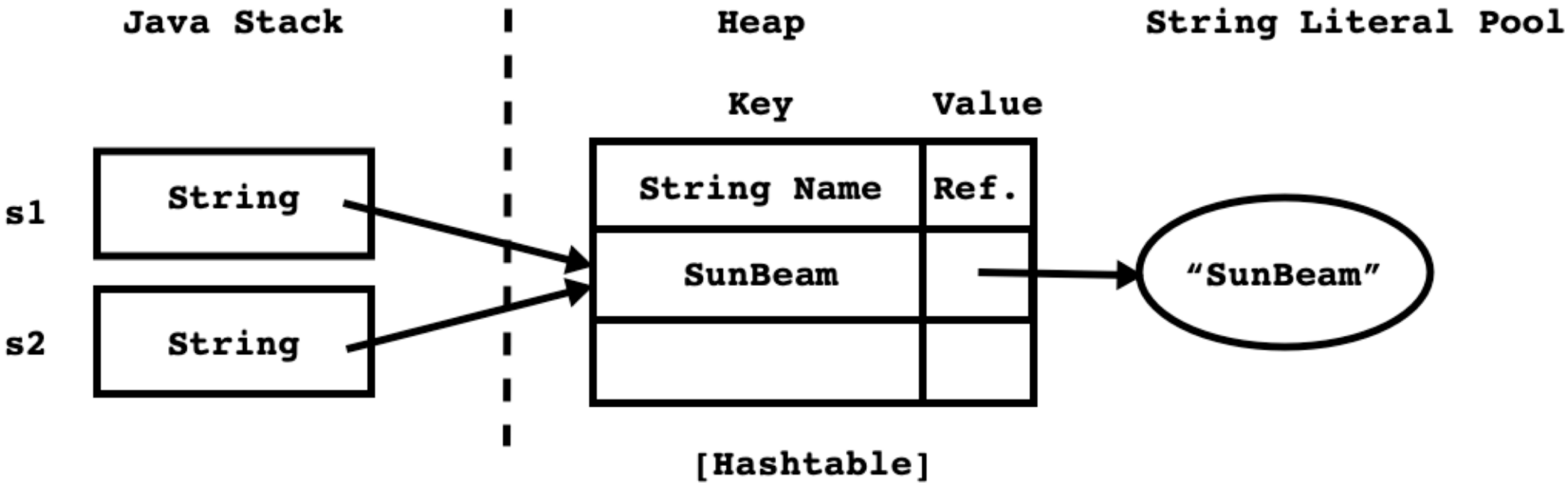
```
public class Program {
   public static void main(String[] args) {
      String s1 = "SunBeam";
      String s2 = "SunBeam";
      if( s1.equals(s2) )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Equal
   }
}
```

String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = "SunBeam";
      String s2 = "Sun"+"Beam";
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Equal
   }
}
```

String Twister

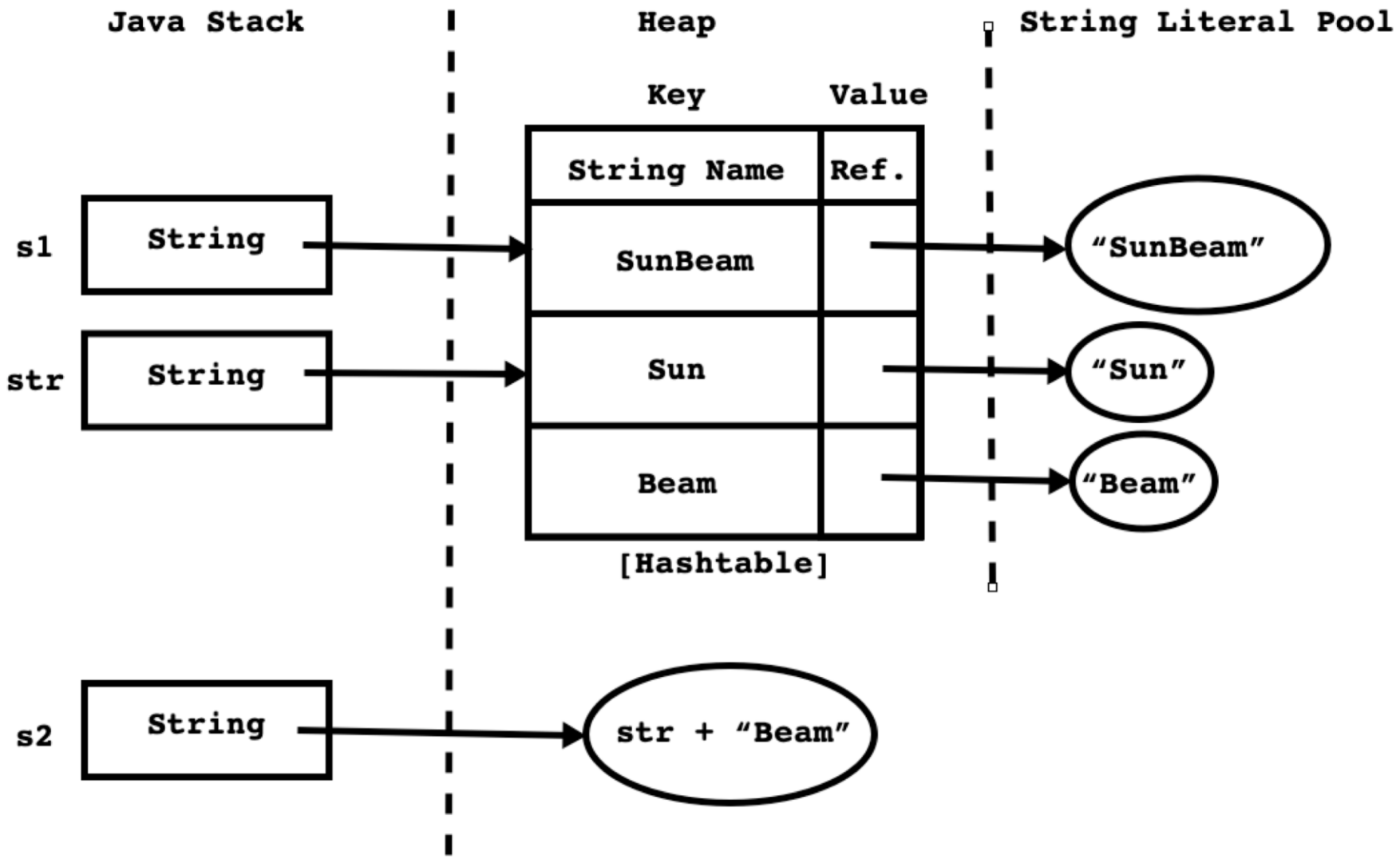
- Constant expression gets evaluated at compile time.
  - "int result = 2 + 3;" becomes "int result = 5;" at compile time
  - "String s2 = "Sun"+"Beam";" becomes "String s2="SunBeam";" at compile time.



String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = "SunBeam";
      String str = "Sun";
      String s2 = str + "Beam";
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

String Twister



String Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = "SunBeam";
      String str = "Sun";
      String s2 = ( str + "Beam" ).intern();
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Equal
   }
}
```

String Twister

```
public class Program {
   public static final String str = "Hello";
   public static void main(String[] args) {
      String str = "Hello";
      System.out.println(A.str == B.str);          //true
      System.out.println(A.str == Program.str);  //true
      System.out.println(A.str == str);          //true
      System.out.println(B.str == Program.str);  //true
      System.out.println(B.str == str);          //true
      System.out.println(Program.str == str);    //true
   }
}
```

String Twister

```
public class Program {
   public static void main(String[] args) {
      String str = "SunBeam";
      //char ch = str.charAt( 0 );   //S
      //char ch = str.charAt( 6 );   //m
      //char ch = str.charAt(-1); //StringIndexOutOfBoundsException
      char ch = str.charAt( str.length() ); //StringIndexOutOfBoundsException
      System.out.println(ch);
   }
}
```

StringBuffer versus StringBuilder

- StringBuffer and StringBuilder are final classes.
- It is declared in java.lang package.
- It is used create to mutable string instance.
- equals() and hashCode() method is not overridden inside it.
- We can create instances of these classes using new operator only.
- Instances get space on Heap.
- **StringBuffer implementation is thread safe whereas StringBuilder is not.**
- **StringBuffer is introduced in JDK1.0and StringBuilder is introduced in JDK 1.5.**

StringBuffer Twister

```
public class Program {
   public static void main(String[] args) {
      StringBuffer s1 = new StringBuffer("SunBeam");
      StringBuffer s2 = new StringBuffer("SunBeam");
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

StringBuffer Twister

```
public class Program {
   public static void main(String[] args) {
      StringBuffer s1 = new StringBuffer("SunBeam");
      StringBuffer s2 = new StringBuffer("SunBeam");
      if( s1.equals(s2) )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

StringBuffer Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = new String("SunBeam");
      StringBuffer s2 = new StringBuffer("SunBeam");
      if( s1 == s2 )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Compiler Error
   }
}
```

StringBuffer Twister

```
public class Program {
   public static void main(String[] args) {
      String s1 = new String("SunBeam");
      StringBuffer s2 = new StringBuffer("SunBeam");
      if( s1.equals(s2) )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

StringBuffer Twister

```
public class Program {
   public static void main(String[] args) {
      StringBuffer s1 = new StringBuffer("SunBeam");
      String s2 = new String("SunBeam");
      if( s1.equals(s2) )
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

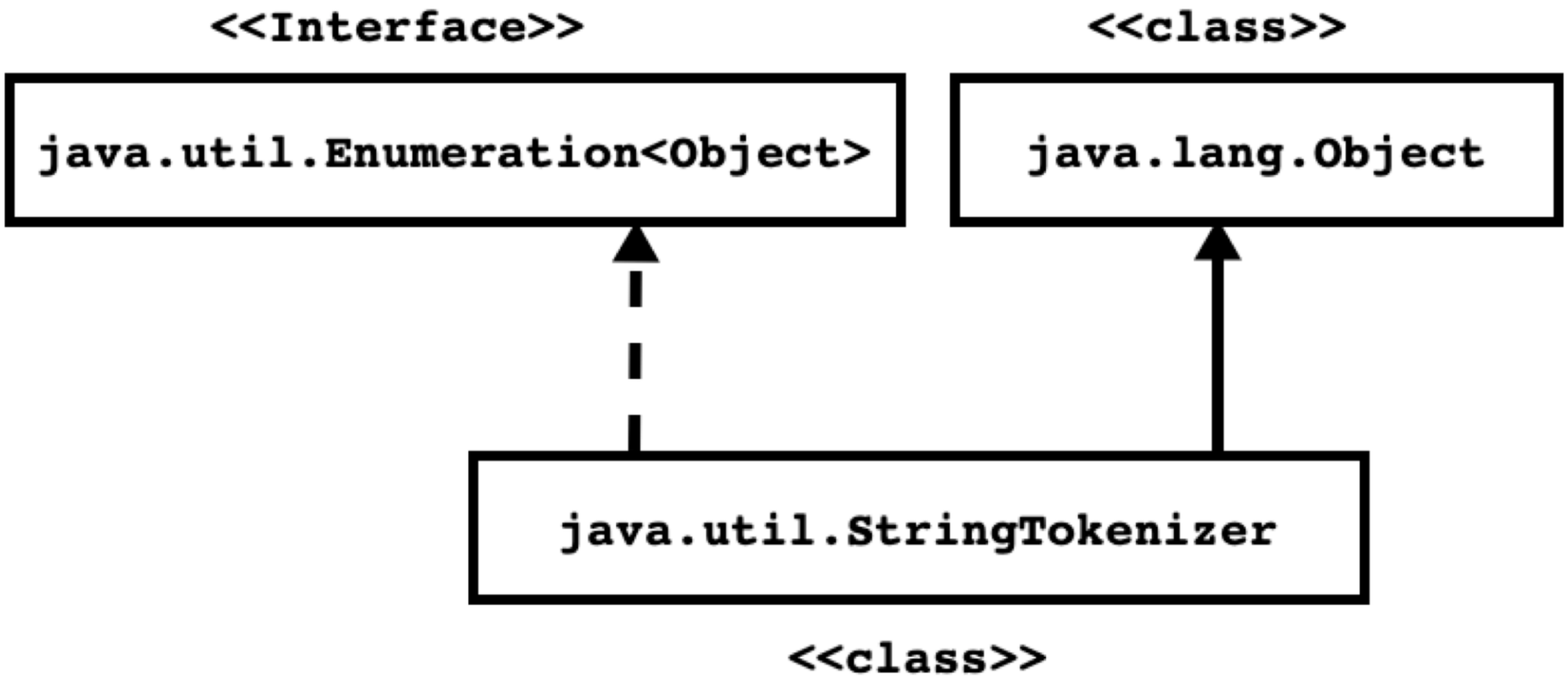
StringBuilder Twister

```
public class Program {
   public static void main(String[] args) {
      StringBuilder s1 = new StringBuilder("SunBeam");
      StringBuilder s2 = new StringBuilder("SunBeam");
      if( s1 ==  s2)
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

StringBuilder Twister

```
public class Program {
   public static void main(String[] args) {
      StringBuilder s1 = new StringBuilder("SunBeam");
      StringBuilder s2 = new StringBuilder("SunBeam");
      if( s1.equals(s2))
         System.out.println("Equal");
      else
         System.out.println("Not Equal");
      //Output : Not Equal
   }
}
```

StringTokenizer



StringTokenizer

- The string tokenizer class allows an application to break a string into tokens.
- Methods of java.util.Enumeration<E> interface
  1. boolean hasMoreElements( )
  2. E nextElement()
- Methods of java.util.StringTokenizer class
  1. public int countTokens()
  2. public boolean hasMoreTokens()
  3. public String nextToken()
  4. public String nextToken(String delim)

StringTokenizer

```
public class Program {
   public static void main(String[] args) {
      String str = "www.sunbeaminfo.com";
      String delim = ".";
      StringTokenizer stk = new StringTokenizer(str, delim);
      String token = null;
      while( stk.hasMoreTokens()) {
         token = stk.nextToken();
         System.out.println(token);
      }
   }
```

```
Output
------------
www
sunbeaminfo
com
```

Generic Programming(Using Generics)

```
class Box<T>{    //T : Type Parameter
    private T object;
    public T getObject() {
        return object;
    }
    public void setObject(T object) {
        this.object = object;
    }
}
```

```
Box<Date> b1 = new Box<Date>( );    //Date : Type Argument

Box<Date> b1 = new Box<>( ); //Type Inference
```

- By passing, data type as a argument, we can write generic code in Java. Hence parameterized class/type is called as generics.
- If we specify type argument during declaration of reference variable then specifying type argument during instance creation is optional. It is also called as type inference.

Generic Programming(Using Generics)

```
1. Box<String> b1 = new Box<String>( ); //OK

2. Box<String> b2 = new Box<>( ); //OK

3. Box<Object> b3 = new Box<String>( ); //NOT OK

4. Box<Object> b4 = new Box<Object>( ); //OK
```

Generic Programming(Using Generics)

- If we instantiate parameterized type without type argument then java.lang.Object is considered as default type argument.

```
Box b1 = new Box( );     //Class Box : Raw Type
//Box<Object> b1 = new Box<>( );
```

- If we instantiate parameterized type without type argument then parameterized type is called raw type.

- During the instantiation of parameterized class, type argument must be reference type.

```
//Box<int> b1 = new Box< >( );   //Not OK
Box<Integer> b1 = new Box< >( );     //OK
```

Why Generics?

1. It gives us stronger type checking at compile time. In other words, we can write type safe code.
2. No need of explicit type casting.
3. We can implement generic data structure and algorithm.

Type Parameters Used In Java API

1. T : Type
2. N : Number
3. E : Element
4. K : Key
5. V : Value
6. S, U : Second Type Parameter Names

Restrictions on Generics

1. Cannot Instantiate Generic Types with Primitive Types

2. Cannot Create Instances of Type Parameters

3. Cannot Declare Static Fields Whose Types are Type Parameters

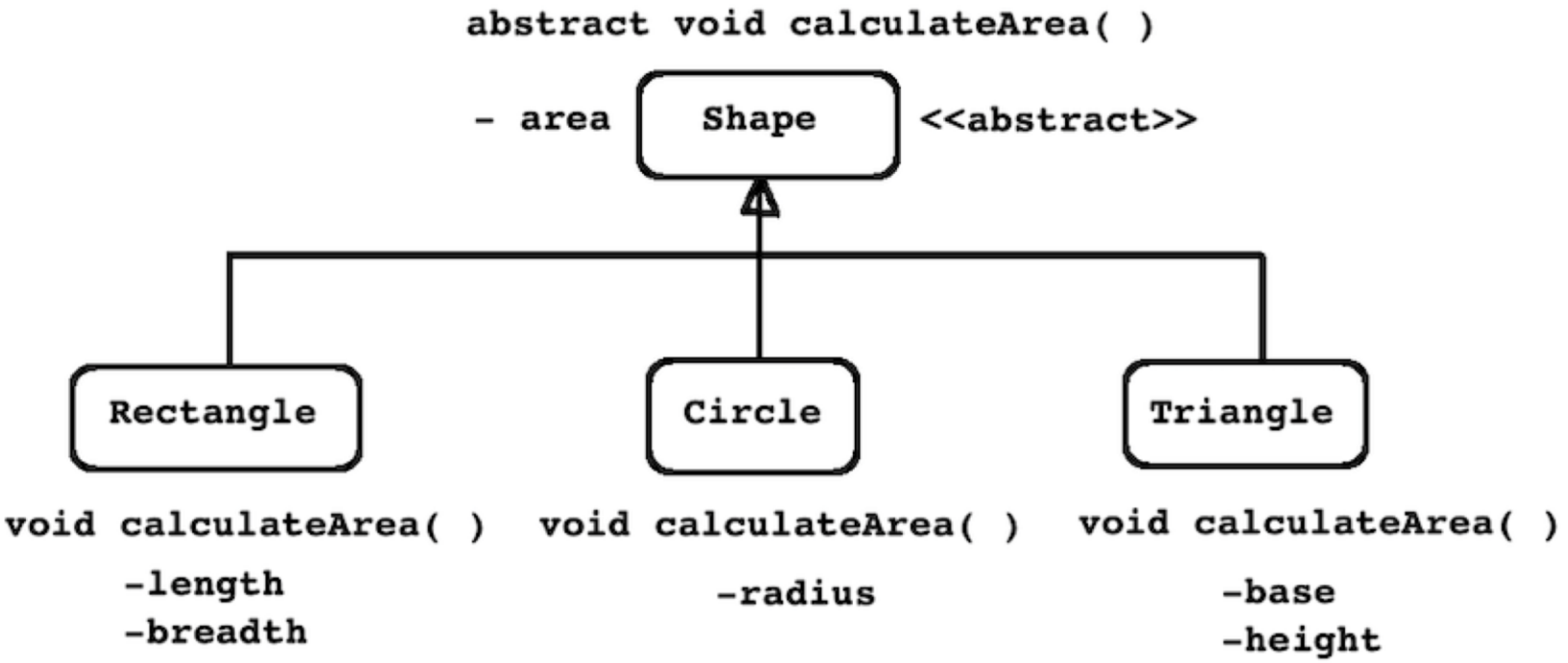
4. Cannot Use Casts or instanceof with Parameterized Types

5. Cannot Create Arrays of Parameterized Types

6. Cannot Create, Catch, or Throw Objects of Parameterized Types

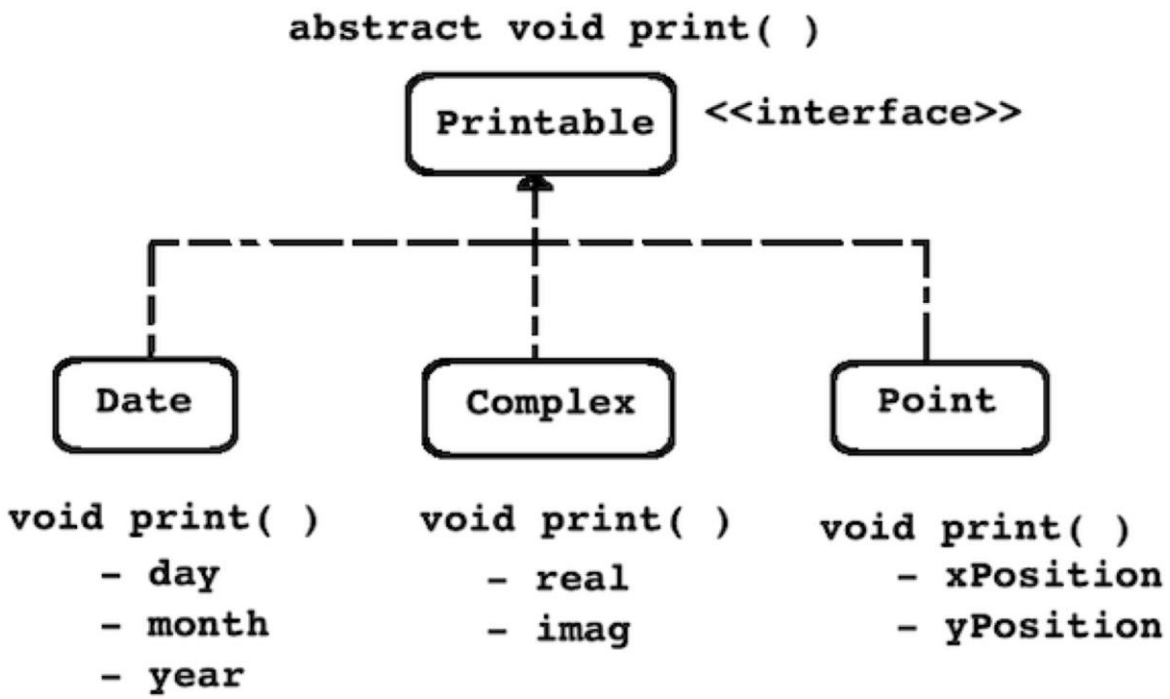
7. Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Abstract Class




```
Shape[] arr = new Shape[ 3 ];
arr[ 0 ] = new Rectangle( ); //Upcasting
arr[ 1 ] = new Circle( ); //Upcasting
arr[ 2 ] = new Triangle( ); //Upcasting
```

Interface




```
Printable[] arr = new Printable[ 3 ];
arr[ 0 ] = new Date( ); //Upcasting
arr[ 1 ] = new Complex( ); //Upcasting
arr[ 2 ] = new Point( ); //Upcasting
```

Commonly Used Interfaces

1. java.lang.AutoCloseable
2. java.io.Closeable
3. java.lang.Cloneable
4. java.lang.Comparable
5. java.util.Comparator
6. java.lang.Iterable
7. java.util.Iterator
8. java.io.Serializable

Comparable

- It is interface declared in java.lang package.
- **"int compareTo(T other)"** is a method of java.lang.Comparable interface.
- If state of current object is less than state of other object then compareTo() method should return negative integer(-1).
- If state of current object is greater than state of other object then compareTo() method should return positive integer(+1).
- If state of current object is equal to state of other object then compareTo() method should return zero(0).
- If we want to sort, array of non primitive type which contains all the instances of same type then we should implement Comparable interface.

Comparator

- It is interface declared in java.util package.
- **"int compare(T o1, T o2)"** is a method of java.util.Comparator interface.
- If state of current object is less than state of other object then compare() method should return negative integer(-1).
- If state of current object is greater than state of other object then compare() method should return positive integer(+1).
- If state of current object is equal to state of other object then compare() method should return zero(0).
- If we want to sort, array of instances of non primitive of different type then we should implement Comparator interface.

Iterable and Iterator Implementation

- Iterable<T> is interface declared in java.lang package.
- Implementing this interface allows an object to be the target of the "for-each loop" statement.
- It is introduced in JDK 1.5
- Methods of java.lang.Iterable interface:
    1. Iterator<T> iterator()
    2. default Spliterator<T> spliterator()
    3. default void forEach(Consumer<? super T> action)

Iterable and Iterator Implementation

- Iterator<E> is interface declared in java.util package.
- It is used to traverse collection in forward direction only.
- It is introduced in JDK 1.2
- Methods of java.util.Iterator interface:
    1. boolean hasNext()
    2. E next()
    3. default void remove()
    4. default void forEachRemaining(Consumer<? super E> action)

Iterable and Iterator Implementation

```
LinkedList<Integer> list = new LinkedList<>( );
    list.add(10);
    list.add(20);
    list.add(30);

    for( Integer e : list )
        System.out.println(e);
```

```
foreach loop implicitly work as follows
Integer element = null;
Iterator<Integer> itr = list.iterator();
while( itr.hasNext()) {
    element = itr.next();
    System.out.println(element);
}
```

Thank You.