Object as a Universal Super Class and Wrapper Classes

Object class

- Object is a universal superclass in java
- The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.
- The Object class is beneficial if you want to refer any object whose type you don't know.

  Notice that parent class reference variable can refer the child class object, known as upcasting.

Object class

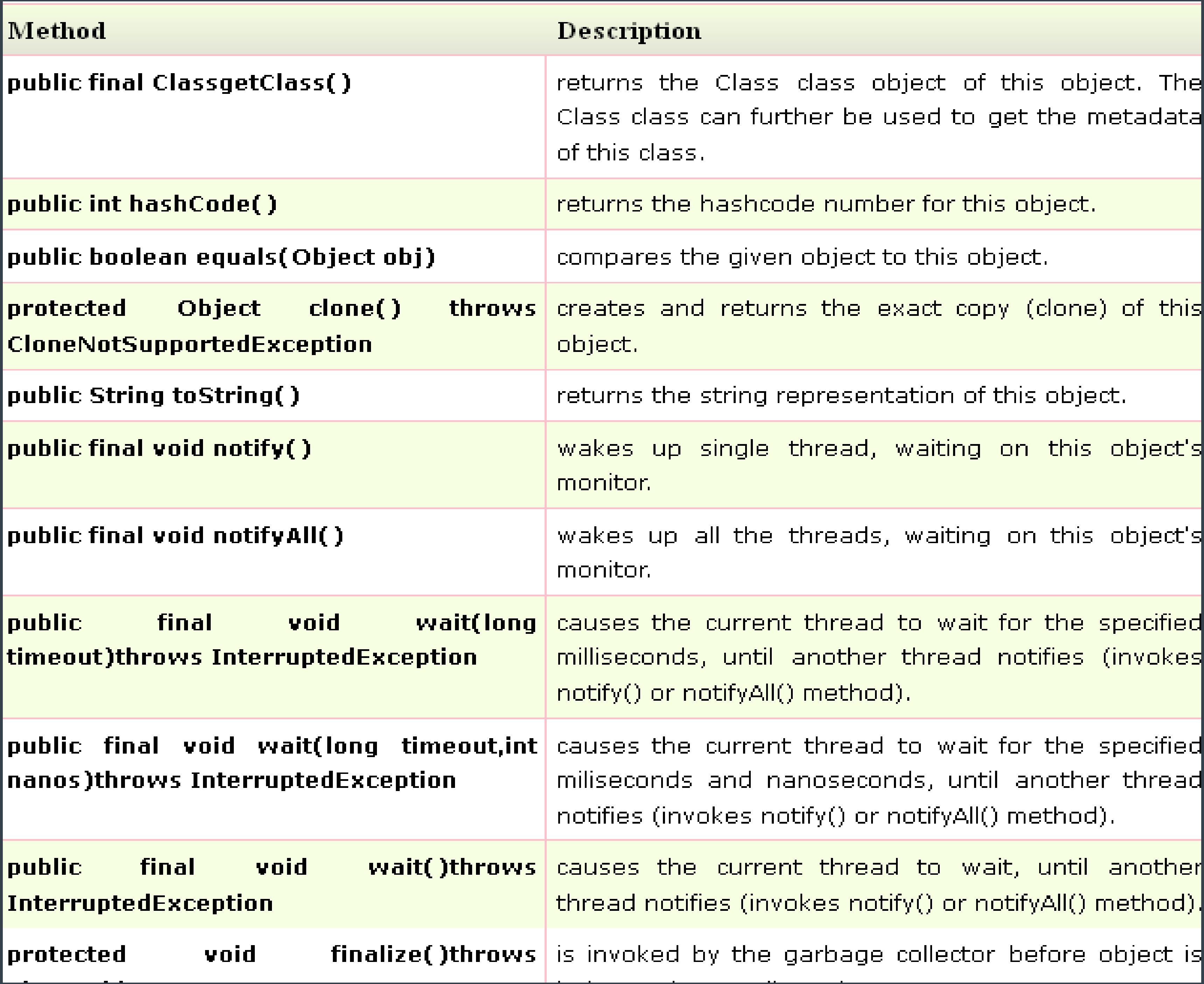
- E.g., there is getObject() method that returns an object but it can be of any type like Employee, Student etc, we can use Object class reference to refer that object. For example:
- Object obj = getObject();

    we don't know
    what object would be returned from this method

- The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.

Con’t

- Of course, a variable of type Object is only useful as a generic holder for arbitrary values.
- If you have some knowledge about the original type and then apply a cast:
- Employee e = (Employee) obj;
- Only primitive types (numbers, characters, and Boolean values) are not objects.
- All array types no matter whether they are arrays of objects or arrays of primitive types, are class types that extent the object class.



Method	Description
public final ClassgetClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws	is invoked by the garbage collector before object is

java.lang.Object

- The Java super class **java.lang.Object** has two very important methods defined in it. They are -

  public boolean equals(Object obj)

  public int hashCode()

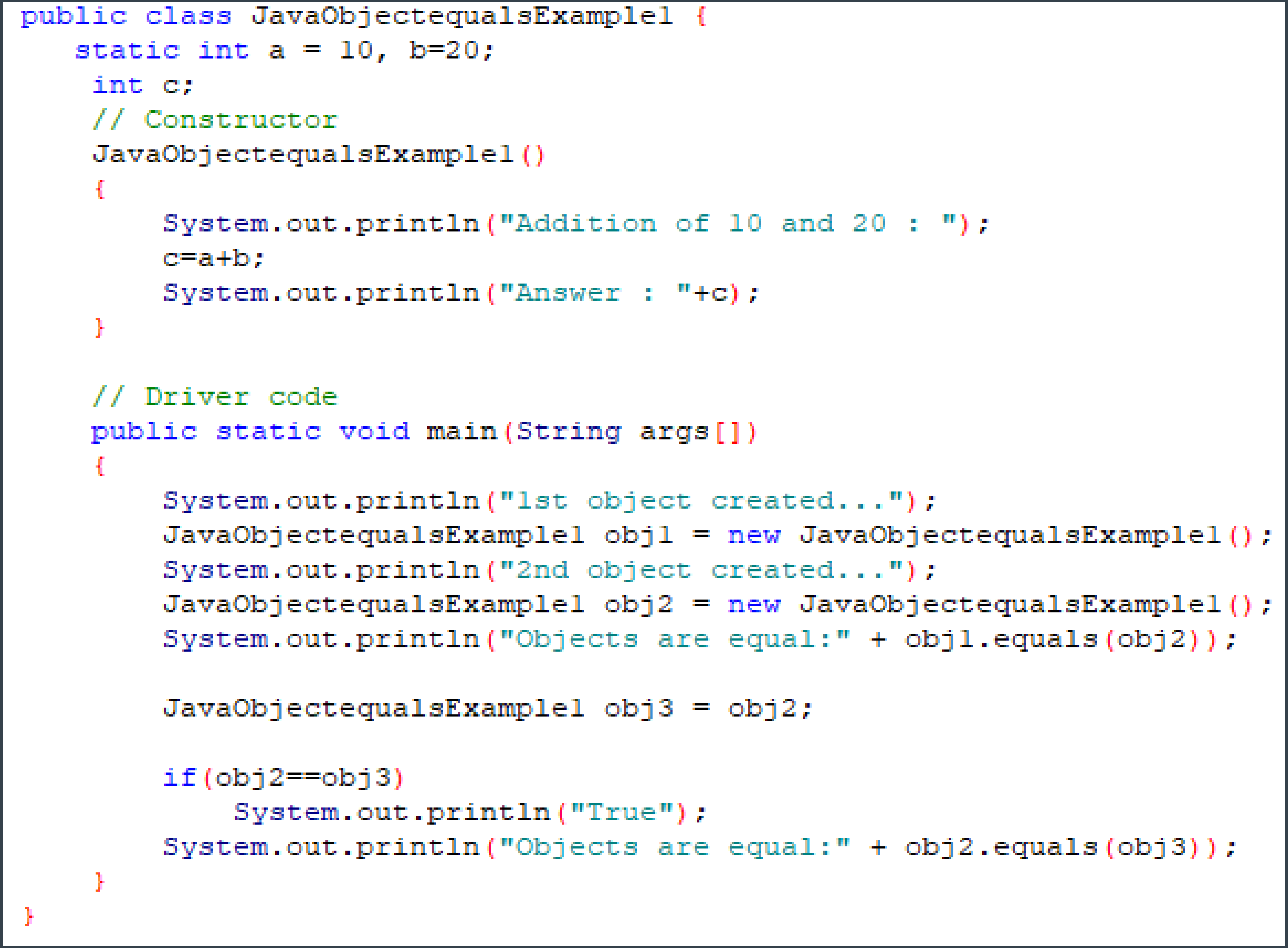
- The equals() method checks if some other object passed to it as an argument is *equal* to the object on which this method is invoked.
- The default implementation of this method in Object class simply checks if two object references x and y refer to the same object.

  i.e. It checks if x == y.

public Boolean equals(Object obj)

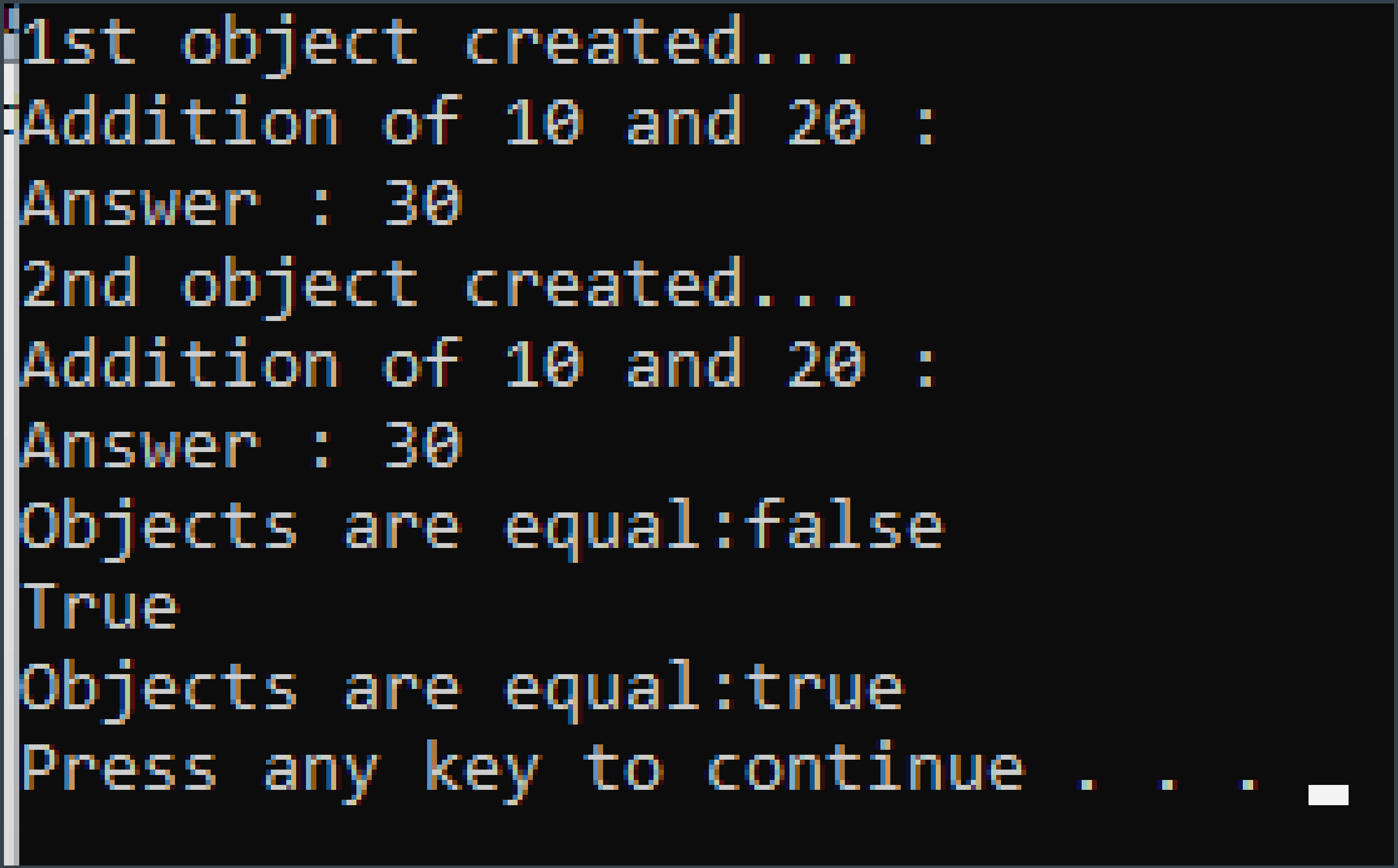
- Object class has no data members that define its state, So, it simply performs comparison.
- However, the classes providing their own implementations of the equals method are supposed to perform a "deep comparison"
- The equals method for class Object implements the most discriminating possible equivalence relation on objects;

  that is, for any reference values x and y, this method returns true if and only if x and y refer to the same object.



```java
public class JavaObjectequalsExample1 {
   static int a = 10, b=20;
    int c;
    // Constructor
    JavaObjectequalsExample1()
    {
        System.out.println("Addition of 10 and 20 : ");
        c=a+b;
        System.out.println("Answer : "+c);
    }

    // Driver code
    public static void main(String args[])
    {
        System.out.println("1st object created...");
        JavaObjectequalsExample1 obj1 = new JavaObjectequalsExample1();
        System.out.println("2nd object created...");
        JavaObjectequalsExample1 obj2 = new JavaObjectequalsExample1();
        System.out.println("Objects are equal:" + obj1.equals(obj2));

        JavaObjectequalsExample1 obj3 = obj2;

        if(obj2==obj3)
            System.out.println("True");
        System.out.println("Objects are equal:" + obj2.equals(obj3));
    }
}
```

public Boolean equals(Object obj)



```
1st object created...
Addition of 10 and 20 :
Answer : 30
2nd object created...
Addition of 10 and 20 :
Answer : 30
Objects are equal:false
True
Objects are equal:true
Press any key to continue . . .
```

public int hashCode()

- This method returns the hash code value for the object on which this method is invoked.
- It returns the hash code value as an integer and is supported for the benefit of hashing based collection classes such as Hashtable, HashMap, HashSet etc.
- This method must be overridden in every class that overrides the equals method.

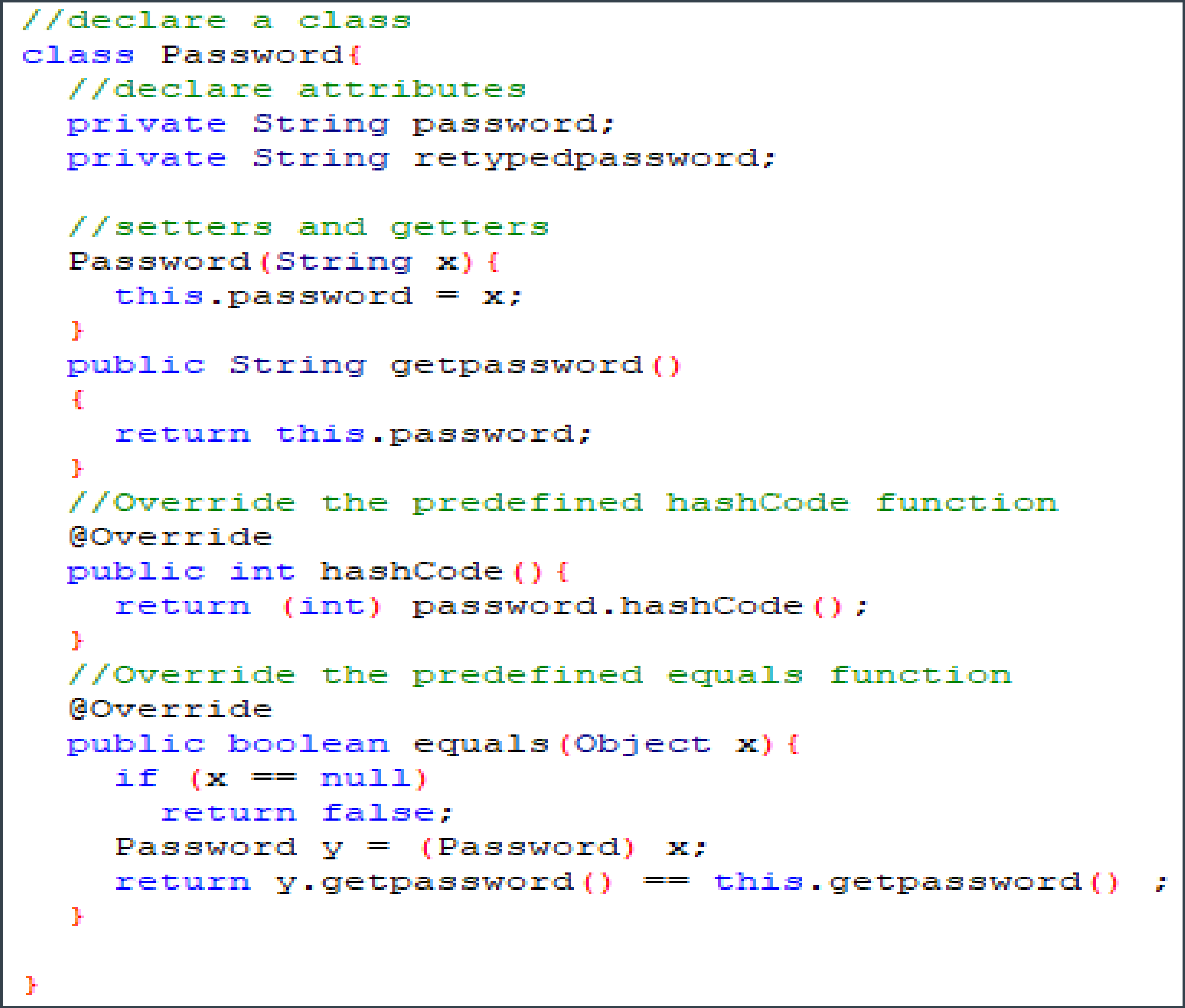
The general contract of hashCode

- Whenever it is invoked on the same object more than once during an execution of a Java application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified.

    This integer need not remain consistent from one execution of an application to another execution of the same application.

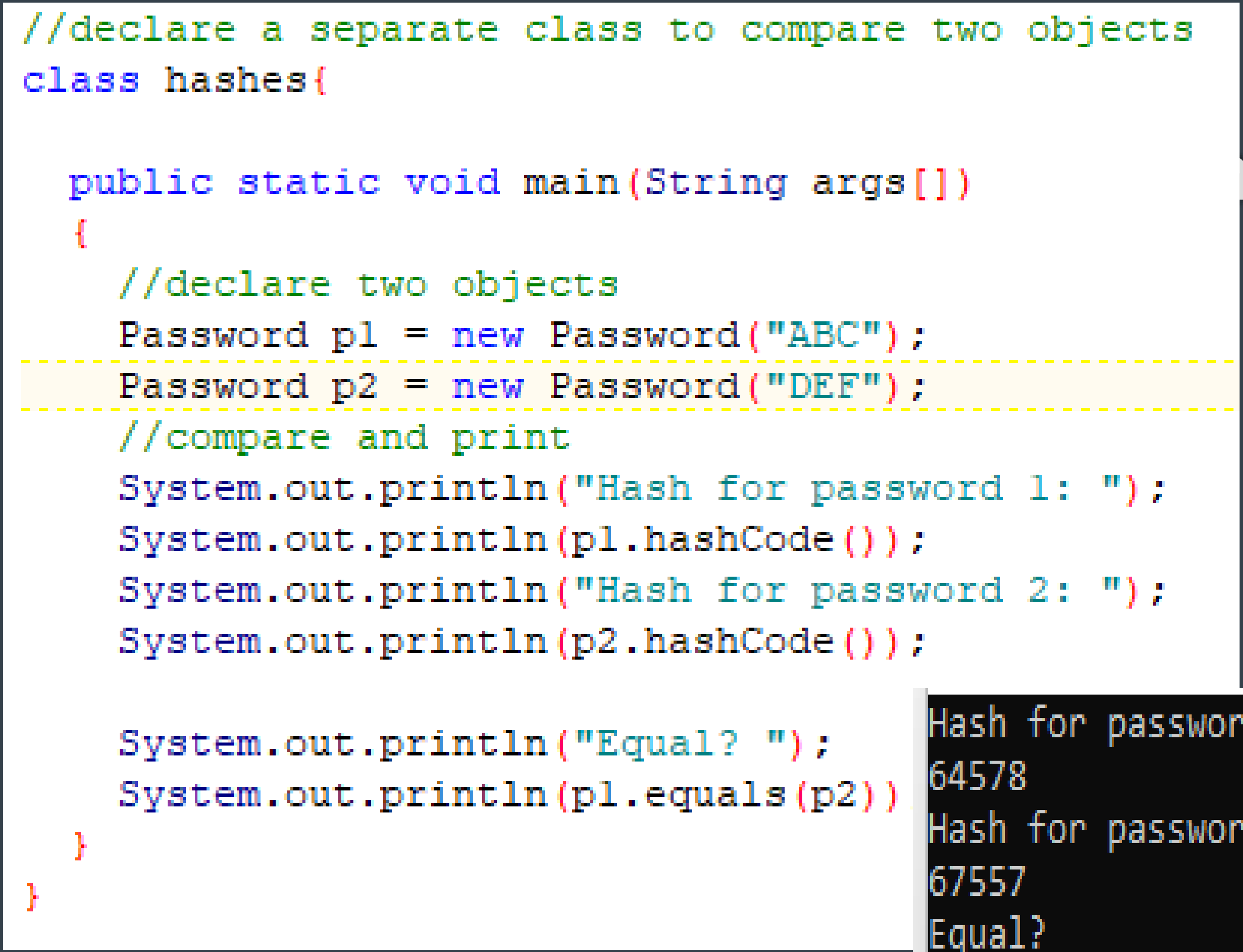
The general contract of hashCode

- If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the equals(java.lang.Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results.

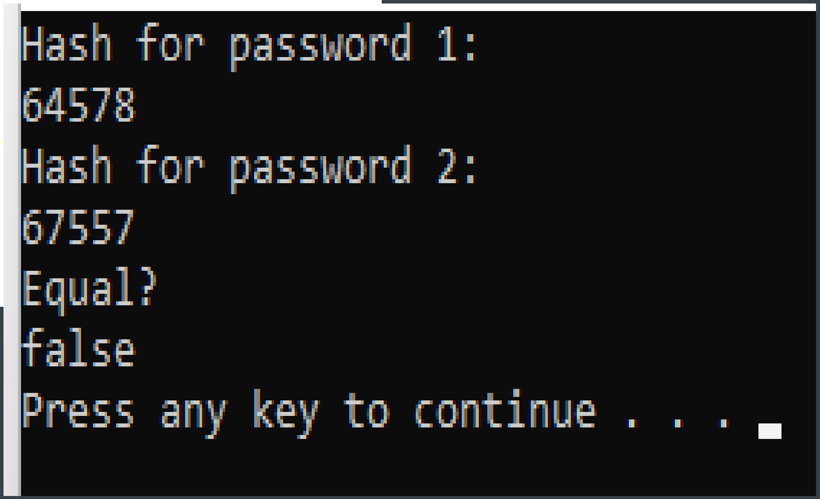


```
//declare a class
class Password{
   //declare attributes
   private String password;
   private String retypedpassword;

   //setters and getters
   Password(String x){
      this.password = x;
   }
   public String getpassword()
   {
      return this.password;
   }
   //Override the predefined hashCode function
   @Override
   public int hashCode(){
      return (int) password.hashCode();
   }
   //Override the predefined equals function
   @Override
   public boolean equals(Object x){
      if (x == null)
         return false;
      Password y = (Password) x;
      return y.getpassword() == this.getpassword() ;
   }

}
```

Code



```java
//declare a separate class to compare two objects
class hashes{

  public static void main(String args[])
  {
    //declare two objects
    Password p1 = new Password("ABC");
    Password p2 = new Password("DEF");
    //compare and print
    System.out.println("Hash for password 1: ");
    System.out.println(p1.hashCode());
    System.out.println("Hash for password 2: ");
    System.out.println(p2.hashCode());

    System.out.println("Equal? ");
    System.out.println(p1.equals(p2))
  }
}
```



```
Hash for password 1:
64578
Hash for password 2:
67557
Equal?
false
Press any key to continue . . .
```

The toString Method

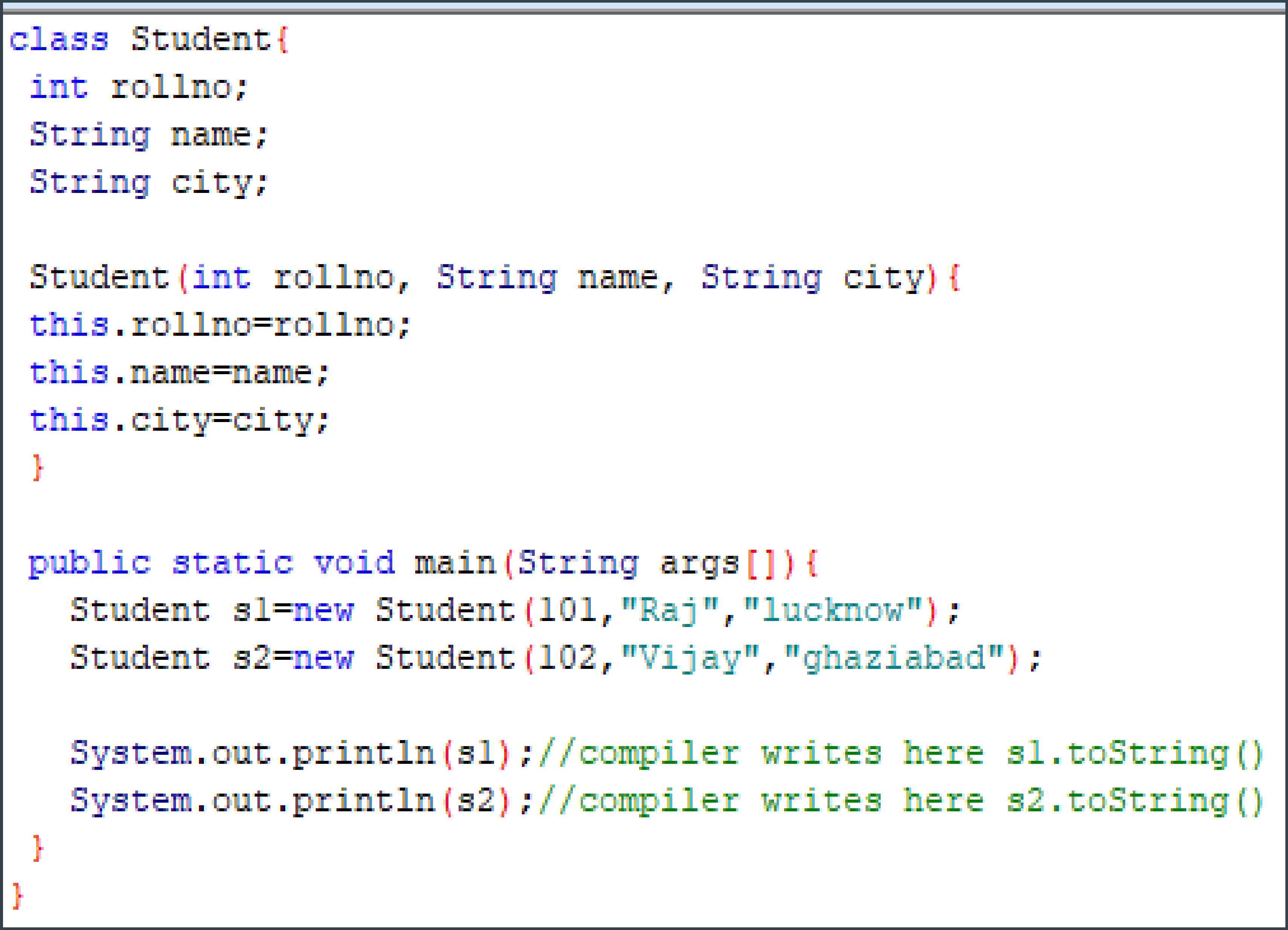
If you want to represent any object as a string, toString() method comes into existence.

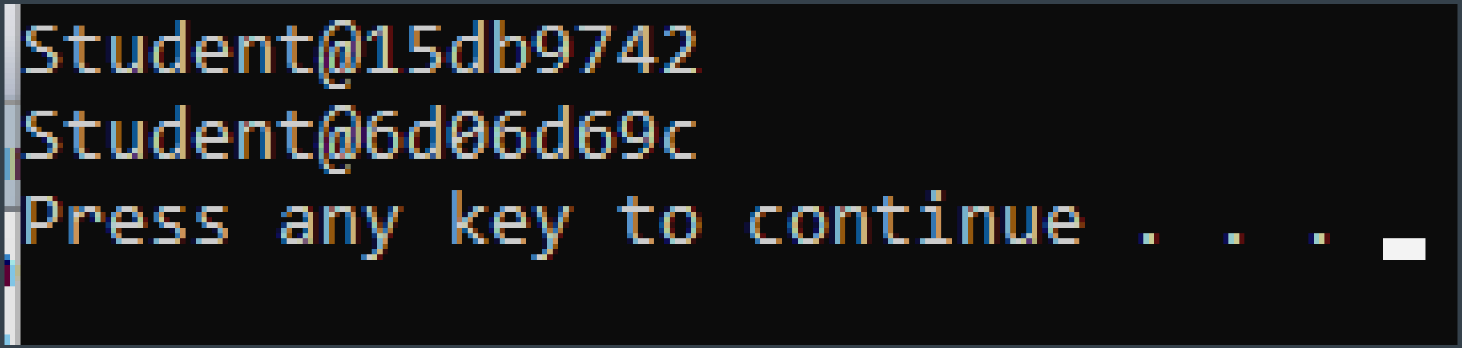
The toString() method returns the String representation of the object.

If you print any object, Java compiler internally invokes the toString() method on the object. So overriding the toString() method, returns the desired output, it can be the state of an object etc. depending on your implementation.

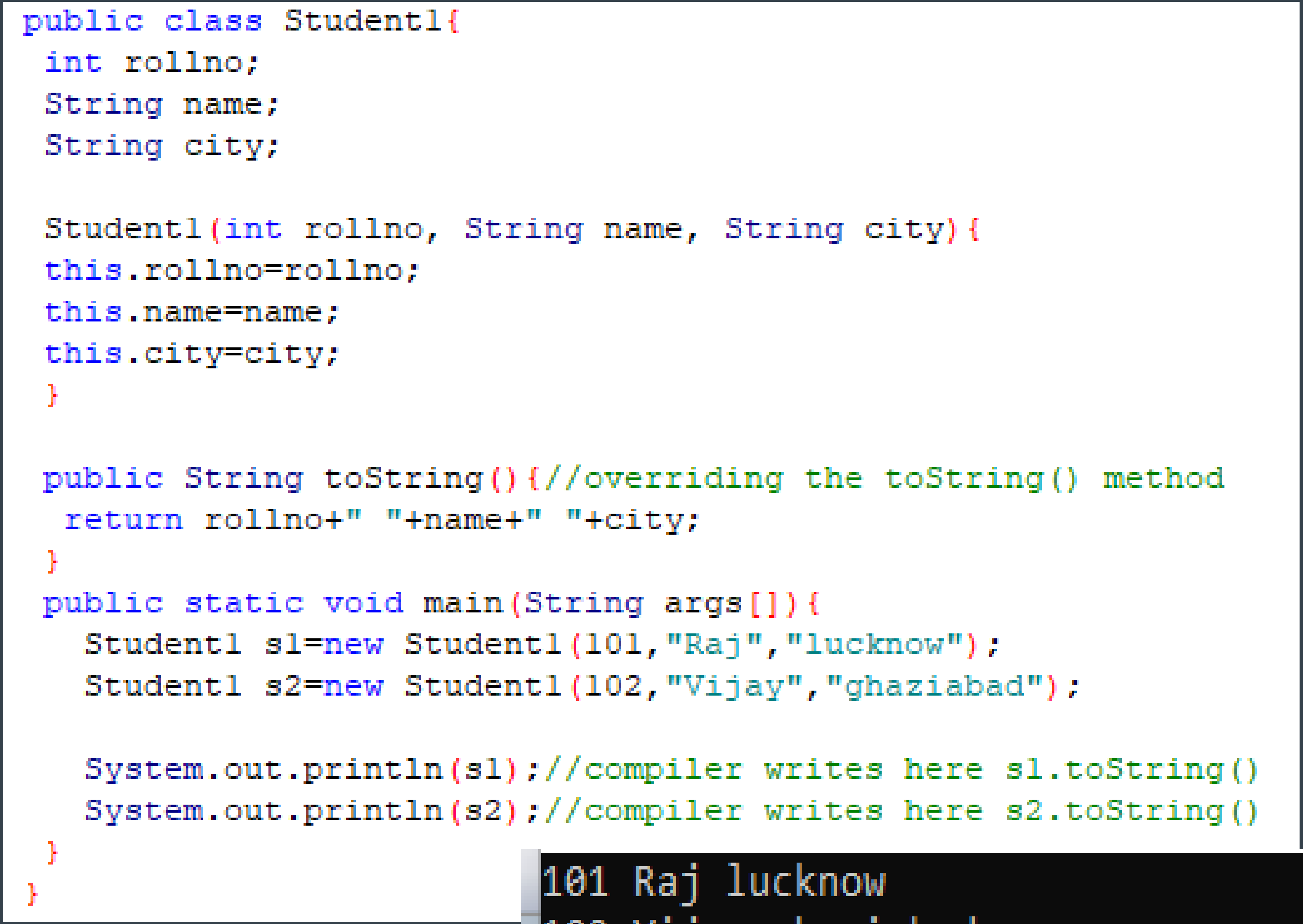
Advantage of Java toString() method

By overriding the toString() method of the Object class, we can return values of the object, so we don't need to write much code.

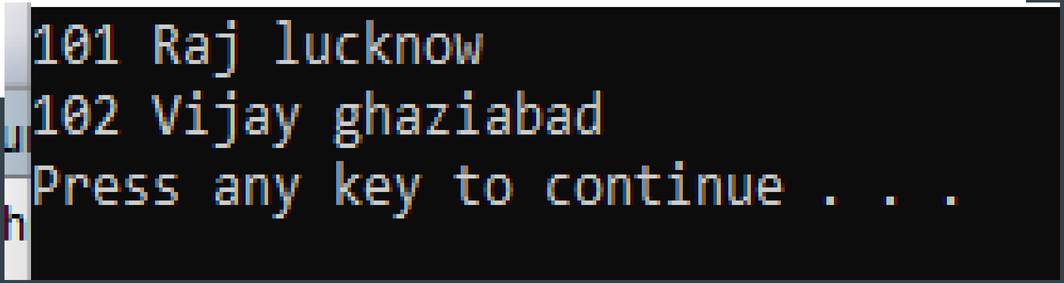

```
class Student{
 int rollno;
 String name;
 String city;

 Student(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public static void main(String args[]){
   Student s1=new Student(101,"Raj","lucknow");
   Student s2=new Student(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```



```
Student@15db9742
Student@6d06d69c
Press any key to continue . . .
```



```java
public class Student1{
 int rollno;
 String name;
 String city;

 Student1(int rollno, String name, String city){
 this.rollno=rollno;
 this.name=name;
 this.city=city;
 }

 public String toString(){//overriding the toString() method
  return rollno+" "+name+" "+city;
 }
 public static void main(String args[]){
   Student1 s1=new Student1(101,"Raj","lucknow");
   Student1 s2=new Student1(102,"Vijay","ghaziabad");

   System.out.println(s1);//compiler writes here s1.toString()
   System.out.println(s2);//compiler writes here s2.toString()
 }
}
```



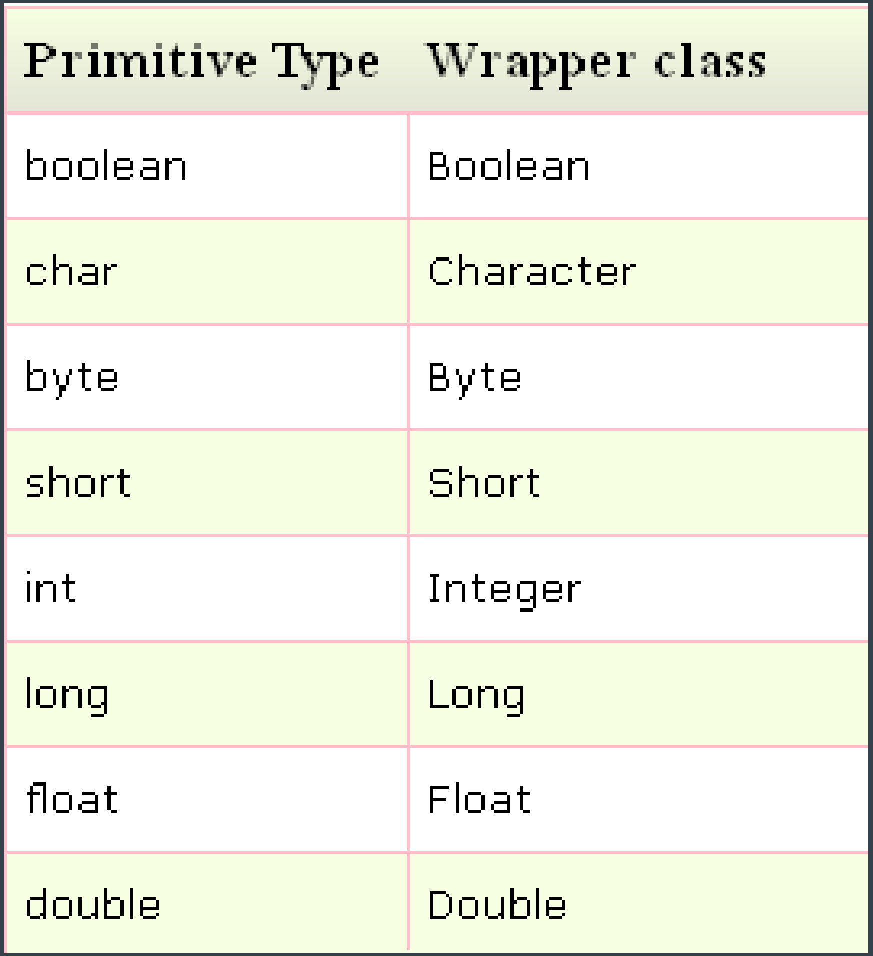
```
101 Raj lucknow
102 Vijay ghaziabad
Press any key to continue . . .
```

Wrapper Classes

- Wrapper class in java provides the mechanism to convert primitive into object and object into primitive.
- Since J2SE 5.0, autoboxing and unboxing feature converts primitive into object and object into primitive automatically.
- The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing.
- **Why we need Wrapper Class?**
- Wrapper Class will convert primitive data types into objects. The objects are necessary if we wish to modify the arguments passed into the method (because primitive types are passed by value).

Wrapper Classes

- The list of eight wrapper classes are given below:



Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper class hierarchy

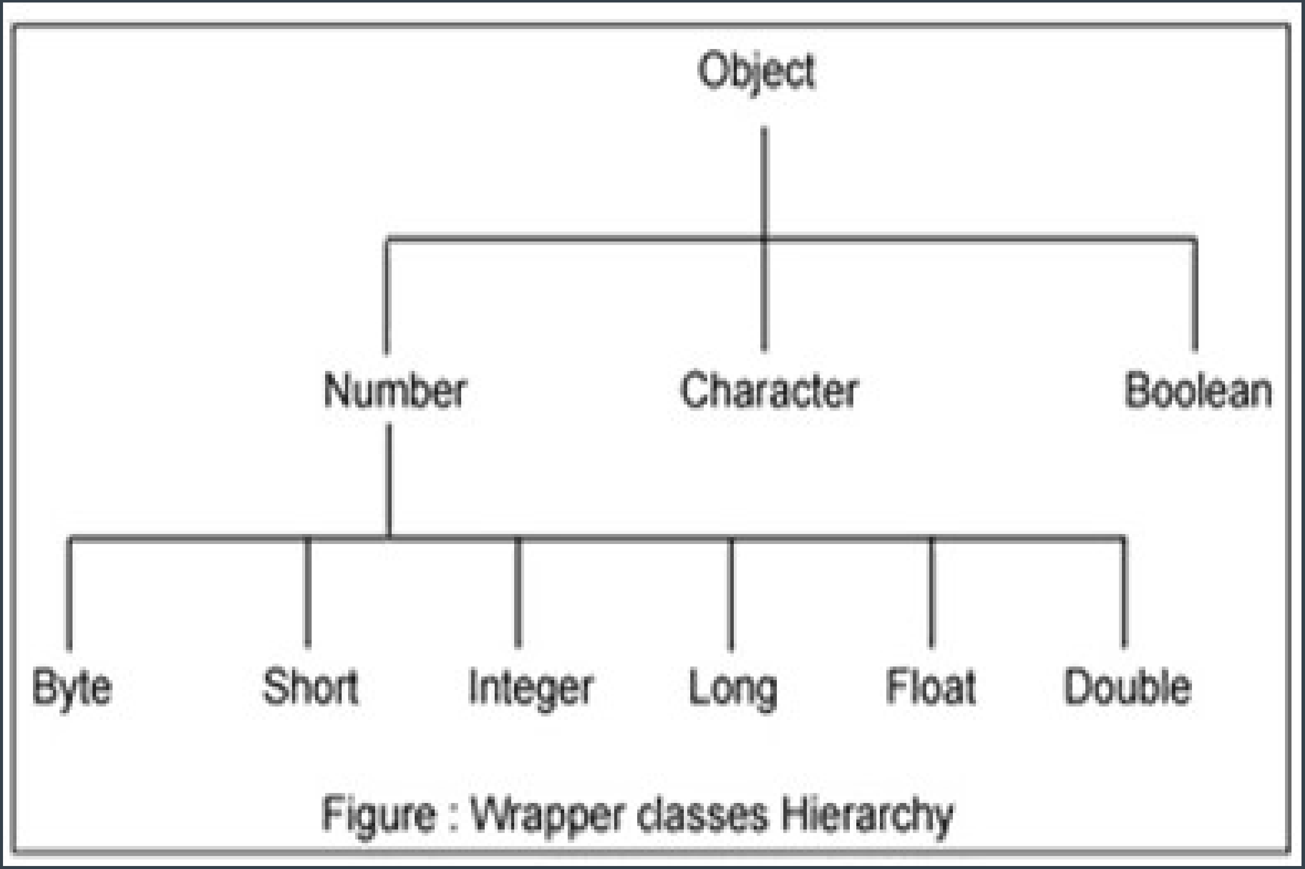


Figure : Wrapper classes Hierarchy

Wrapper class Example: Primitive to Wrapper

- public class WrapperExample1{

```
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf (a); //converting int into Integer
//autoboxing, now compiler will write Integer.valueOf
   //(a) internally
Integer j=a;
  System.out.println (a+" "+i+" "+j);
}
```

}

Wrapper class Example: Wrapper to Primitive

```
public class WrapperExample2{
 public static void main(String args[]){
     //Converting Integer to int
     Integer a=new Integer(3);
     int i=a.intValue();//converting Integer to int
     //unboxing, now compiler will write a.intValue() internally
     int j=a;
     System.out.println(a+" "+i+" "+j);
 }
}
```

Wrapper and Primitive

Wrapper class provides a mechanism to convert primitive type into object and object into primitive type.
A primitive type is a predefined data type provided by Java.

A Wrapper class is used to create an object; therefore, it has a corresponding class.
A Primitive type is not an object so it does not belong to a class.

Required memory is higher than the primitive types.
Required memory is lower comparing to wrapper classes.