

```
Employee c = new Employee("Sally Brown");

System.out.println(Employee.getNextID());

System.out.println(a.getID() + " : " + a.getName());
System.out.println(b.getID() + " : " + b.getName());
System.out.println(c.getID() + " : " + c.getName());
}
}
```

would produce this output:

```
0
3
0 : John Doe
1 : Jane Smith
2 : Sally Brown
```

Creating Objects

Navigate **Classes and Objects** topic:

- Defining classes
- Creating objects
- Interfaces
- Using static members
- Destroying objects
- Overloading methods and constructors
- Class Loading

Introduction

Before a Java object can be created the class byte code must be loaded from the file system (with .class extension) to memory. This process of locating the byte code for a given class name and converting that code into a Java Class class instance is known as class loading. There is one Class created for each type of Java class.

All objects in java programs are created on heap memory. An object is created based on its class. You can consider a class as a blueprint, template, or a description how to create an object. When an object is created, memory is allocated to hold the object properties. An object reference pointing to that memory location is also created. To use the object in the future, that object reference has to be stored as a local variable or as an object member variable.

The Java Virtual Machine (JVM), keeps track of the usage of object references. If there are no more reference to the object, the object can not be used any more and becomes garbage. After a while the heap memory will be full of unused objects. The JVM collects those garbage objects and frees the memory they allocated, so the memory can be reused again when a new object is created. See below a simple example:

```
{
// --- Create an object ---
MyObject obj = new MyObject();

// --- Use the object ---
```

```
obj.printMyValues();  
}
```

The obj contains the object reference pointing to an object created from the MyObject class. The obj object reference is in scope inside the { }. After the } the object becomes garbage. Object references can be passed in to methods, object references can be returned from methods.

Creating object with the new keyword

99% of new objects are created using the **new** keyword.

```
{  
    // --- Create an 'MyObject' for the first time the application  
started --  
    MyObject obj = new MyObject();  
}
```

When an object from the MyObject class is created for the first time. The JVM searches the file system for the definition of the class, that is the Java byte code. The file has the extension of '*.class'. The **CLASSPATH** environment variable contains locations where Java classes are stored. The JVM is looking for the 'MyObject.class' file. Depending on which package the class belongs to, the package name will be translated to a directory path.

When the 'MyObject.class' file is found, the JVM's class loader loads the class in memory, and creates a Class object. The JVM stores the code in memory, allocates memory for the **static** variables, and executes any static initialize block. Memory is not allocated for the object member variables at this point, memory will be allocated for them when an instance of the class, an object, is created.

There is no limit on how many objects from the same class can be created. Code and **static** variables are stored only once, no matter how many objects are created. Memory is allocated for the object member variables when the object is created. Thus, the size of an object is determined not by its code's size but by the memory it needs for its member variables to be stored.

Creating object by cloning an object

Cloning is not automatically available to classes. There is some help though, as all Java objects inherit the protected Object clone() method. This base method would allocate the memory and do the bit by bit copying of the object's states.

You may ask why we need this clone method. Couldn't I create a constructor and just passing in the same object, and do the copying variable by variable? Lets see:

```
public class MyObject  
{  
    private int memberVar;  
    ...  
    MyObject( MyObject obj )  
    {  
        this.memberVar = obj.memberVar;  
        ...  
    }  
    ...  
}
```

You might think that accessing the private memberVar variable of obj would fail but as this is in the same class this code is legal. The clone() method copies the whole object's memory in one operation. This is much faster than using the new keyword. Object creation with the **new** keyword is expensive, so if you need to create lots of objects with the same type, performance will be better if you create one object and clone new ones from it. See below a factory method that will return a new object using cloning.

```
HashTable _cacheTemplate = new HashTable;
...
/** Clone Customer object for performance reason */
public Customer createCustomerObject()
{
    // --- See if a template object exists in our cache ---
    Customer template = _cacheTemplate.get( "Customer" );
    if ( template == null )
    {
        // --- Create template ---
        template = new Customer();
        _cacheTemplate.put( "Customer", template );
    }
    return template.clone();
}
```

Now, lets see how to make the Customer object cloneable.

- First the Customer class needs to implement the Cloneable Interface.
- Override and make the clone() method **public**, as that is **protected** in the Object class.
- Call the super.clone() method at the beginning of your clone method.
- Override the clone() method in all the subclasses of Customer.

```
public class Customer implements Cloneable
{
    ...
    public Object clone() throws CloneNotSupportedException
    {
        Object obj = super.clone();

        return obj;
    }
}
```

In the above example we used cloning for speed up object creation.

An other use of cloning could be to take a snapshot of an object that can change in time. Lets say we want to store Customer objects in a collection, but we want to disassociate them from the 'live' objects . So before adding the object, we clone them, so if the original object changes from that point forward, the added object won't. Also lets say that the Customer object has a reference to an Activity object that contains the customer activities. Now we are facing a problem, it is not enough to clone the Customer object, we also need to clone the referenced objects. The solution:

- Make the Activity class also cloneable
- Make sure that if the Activity class has other 'changeable' object references, those has to be cloned as well, as seen below