

Lesson 3

Objects and Classes

Knowledge is structured in
layers

WHOLENESS OF THE LESSON

In the OO paradigm of programming, execution of a program involves objects interacting with data and methods. Each object has a type, which is embodied in a Java *class*. The Java language specifies syntax rules for the coding of classes, and also for how objects are to be created based on their type (class). *By using more and more of the intelligence of nature, we are able to successfully manage all complexity in life, and live a life of success, harmony and fulfillment.*

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Template for User-Defined Classes in Java
 - Creating new objects
 - Accessing data and operations in an object
 - Access Modifiers
 - Destroying objects
- Some Classes in the Java Library: `Date`, `GregorianCalendar`, `LocalDate`
- The `'this'` keyword
- Mutators (setters) and Accessors (getters)
- How to Make a Class Immutable
- Boxed Primitives and Autoboxing
- Enum Constants
- Static Fields And Methods
- How Objects and Variables Are Stored in Memory
- Call by Reference vs Call by Value
- Import Packages
- Principles of Good Class Design

The Object-Oriented Paradigm

- The OO paradigm, the elements of the problem domain are viewed as "objects" and then represented within the software design and code as "objects".
- Minimum of translation from real world to machine is required.
- As the programmer, you create software objects that correspond to real-world objects (like Employee class can have an objects of Manager, Director, Labor and need to equip them with the behaviors that the real-world objects actually have (address(), payment(), bonus(), duty()))

The Object-Oriented Paradigm

- Think of every object as providing a set of services, which are specified in its *interface*.
- Analogy: Ignition system on a car – very specific interface (the keyhole) to provide a very specific service (starting the car).
- The automotive engineer figures out how to make the turning of the key/press the button, produce the result of starting the car.
- Likewise, in the world of objects, once the services that an object should provide have been specified, you, as the developer, write the code to ensure that these services are available, and users of your object rely only on the interface to get your object to do things it is supposed to do.

Object-Oriented Concepts

Class – A class defines the properties and behaviors for objects.

A class is a model, template or blueprint from which objects are created. This is the way a particular "type" is created in the Java language such as Customer, Employee, CreditCard, Triangle.

Object construction and instances –objects are instances of a class; the object is a realization of the template. Example: One instance of a Customer class may produce an object representing "Joe Smith"; another instance may represent "Susan Brown"

Object-Oriented Concepts

Encapsulation – Bundling of items together into one entity.

objects in a Java program interact with other objects, by way of their interfaces (list of services (methods)); the data that an object owns and the way that it manages that data are hidden from view (*information hiding*); only the public services provided by the object are visible on the outside. The data and the way it is managed are said to be *encapsulated* in the object.

(instance) fields – the *fields* in a class represents properties(also called attributes) of objects of that class.

Eg: A Customer class might have a *name* field, a *streetAddress* field and a *telephoneNumber* field.

Object-Oriented Concepts

(instance) methods – the *methods* in a class are the behaviors that instances of this class are capable of performing on the data; a Customer class might provide methods *getName*, *updateStreetAddress*, and *lookupTelephoneNumber*

State of an object – the state of an object is the set of values currently stored in its fields

Object-Oriented Concepts

Inheritance – OO languages support the idea that one type is a "subtype" of another; the Triangle type is a subtype of the Shape type. If class B represents a subtype of class A, this relationship can be realized in the language; class B is said to *inherit* from class A; in code, we write `class B extends A`

Fields and methods defined in class A are automatically available to instances of class A. (**Details in Lesson-5**)

Identity – Every object in Java has its own identity. Even if two objects have identical values in their fields, they can be distinguished as different objects.

Template for User-Defined Classes in Java

Anatomy of Java Class:

```
<Modifier> class ClassName
```

```
{
```

```
    // Variables – This represents class attributes
```

```
    // Constructors – To initialize instance variables
```

```
    // Methods - This represents class behavior
```

```
}
```

All the code contained between { } is the class definition

Eg: class Cube {

```
//Empty body for now
```

```
}
```

Defining Instance Variables

- Instance variables are declared using the same syntax as ordinary variables.

- Variables can be prefixed with a visibility modifier

```
modifier data_type variable_name;
```

Example :











```
private int x;  
public float y;  
char c;  
protected double d;
```

To preserve encapsulation, instance variables should be declared *private*.

```
class Cube{  
    int length;  
    int breadth;  
    int height;  
}
```

Variables can have one of 4 different visibilities

Access Control

Modifier	Same class	Same package	Subclass	Universe
private				
default				
protected				
public				

Defining Instance Methods / Behaviour

- Method definitions include a method signature and a method body.
- Methods signatures are defined with the following syntax:

```
modifier return_type method_name(type name, ...)
```

- The return type can be:
 - a fundamental data type
 - an object reference
 - void (no return)
- Parameters are optional
 - If the method takes no parameters, empty brackets are required ()
 - Multiple parameters are separated by commas
 - Parameters are defined by type and name
 - A parameter is a local variable whose scope is in the method.

Defining Your Own Classes-Example

```
class Cube {  
    private int length;  
    private int breadth;  
    private int height;  
    public int getVolume() {  
        return (length * breadth * height);  
    }  
}
```

Declaring And Initializing Objects

- An Object is the manifestation of a class
 - An object is an *Instance* of a class
 - The process of creating an object is called **instantiation**
 - The attributes of an object are called instance variables
 - The methods of an object are called instance methods

Declaring And Initializing Objects

new keyword is used to create an object from a class or by assigning to an already existing object variable

The new operator does two things:

1. it *creates* an instance of a class by allocating memory on heap and
2. it *returns* a reference to that object.

Syntax : `ClassName Object Name = new ClassName();`

```
Cube c1 = new Cube ();
```

```
Cube c2 = c1;
```

Invoking Instance Methods & Variables

- To invoke a method and Variables of an object, use the . (dot) operator

```
objectReference.methodName(parameters)  
objectReference.Variables = Value;
```

If there is a return value, it can be used as an expression

```
c1.getVoulme();
```

Main Point 1

- The OO paradigm is a shift from old design and programming styles which are focused on machine-centric language models. In the OO paradigm, the focus shifts to mapping real world objects and dynamics to software objects and behavior; this parallel structure has proven to be more robust, less error prone, more scalable, and more cost-effective. In SCI we see that a more profound paradigm is discovered when the point of reference moves from the individual to the unbounded level – this is the CC paradigm in which self-sufficiency is based on true knowledge of the Self as universal, rather than the view of the self as a separate individual.

Constructor

- The usual role of a constructor is to initialize the instance variables. The syntax for constructors is similar to that of other methods, with these exceptions:
- The name of the constructor is the same as the name of the class.
- Constructors have no return type and no return value.
- Constructors may accept parameters
- The *default* constructor of any class is the parameter-free constructor
- Constructors cannot be invoked like ordinary methods, but only with new operator.
- One constructor can call another constructor except itself.
- Constructor call should be the first line.

See Demo : lesson3.classexamples.

Destroying Objects: Garbage Collection

- *"Destructors" in languages like C++ . But there are no destructors in Java.*
- *JVM provides a garbage collector.*
 - When objects that have been created during execution of an application are no longer referenced anywhere in the application, they are considered to be *garbage*. Periodically, the JVM will invoke its garbage collector to determine which objects are still being referenced ("mark") and then to free up the memory used up by all the remaining objects ("sweep"). Sweep and Mark is the algorithm used by the garbage collection to identify the unused space and clean up the memory.
- *Sometimes garbage collector is not enough.* Sometimes an unreferenced object ties up more than just a memory location; it may also tie up other resources, like a file or a database connection. In such cases, the garbage collector itself does not know how to – and therefore will not -- free up these resources. To free them up, the developer must write code to handle this need and make sure that it will execute before all references to this object are lost. (Code for this will be discussed in Lesson 12.)
- *Assisting the garbage collector.* To ensure that an object variable no longer refers to a particular object (and to thereby set up the object for garbage collection) it suffices to set the value of the variable to null or to another object.

Dates and Calendars APIs

- Date (Optional)
- GregorianCalendar(Optional)
- LocalDate

Using Predefined Classes(Optional)

Date represents a point in time. It is the number of milliseconds since the beginning of the day 1/1/1970 ("the epoch").

The Date class has a small API:

Two constructors:

Date() //today's date

Date(long numMilliseconds) // another date, passed in the form of millisecs

Methods:

boolean after(Date d)

boolean before(Date d)

Object clone()

int compareTo(Date d)

boolean equals(Date d)

long getTime()

void setTime(long millisecs)

Using Predefined Classes -code

```
public static void main(String[] args) {  
    Date d = new Date(); // Current Date  
    System.out.println(d);  
    Date d1 = new Date(10000000000); // set new Date  
    System.out.println(d1);  
    System.out.println(d.after(d1));    // true  
    System.out.println(d1.after(d));    // false  
    Date d2 = (Date) d1.clone(); // Making a deep copy  
    System.out.println(d1);  
    System.out.println(d.compareTo(d1)); // 1  
    System.out.println(d1.compareTo(d)); // -1  
    System.out.println(d1.equals(d2)); // true  
    System.out.println(d1.equals(d)); // false  
    System.out.println(d.getTime());  
} // lesson3.dateapi\DateExample.java
```

Using Predefined Classes

GregorianCalendar is responsible for calendar operations – calendars are the way one culture represents points in time (examples: Gregorian Calendar, lunar calendar, Mayan calendar) in terms of days, weeks, months, etc.

```
new GregorianCalendar()           // today at this moment  
new GregorianCalendar(1999, 11, 31) // 11 is December
```

- *Conversions:*

```
//get the Date (point in time) represented by cal instance  
GregorianCalendar cal = new GregorianCalendar();  
Date d = cal.getTime();
```

```
//get the GregorianCalendar that corresponds to this Date  
Date d = new Date();  
GregorianCalendar cal = new GregorianCalendar();  
cal.setTime(d);
```

Using Predefined Classes

```
new GregorianCalendar(2023, 11, 31, 23, 12, 58)
new GregorianCalendar(2023, Calendar.DECEMBER, 31)
GregorianCalendar cal = new GregorianCalendar();
int month = cal.get(Calendar.MONTH);           // 0 is January, ...
int weekday = cal.get(Calendar.DAY_OF_WEEK); // 1 - 7
int daynum = cal.get(Calendar.DATE);           // Day of Month : 1, 2, 3, ...
cal.set(Calendar.YEAR, 2024);
cal.set(Calendar.DATE, 23);
// lesson3.dateapi \CalenderExample.java
```

Date and Time Immutable API

- Solves the problem of representing dates and times once and for all, but it is a more complicated API.
- LocalDate manages dates that do not require timezone data, like birthdays and a single-timezone company intranet.
LocalDateTime is like LocalDate but includes time information.
ZonedDateTime, ZonedDateTime handles dates (date and time) and takes into account time zones. They are new and improved versions of GregorianCalendar
- Both Date and Calendar are mutable. All of Java 8 Date and Time classes are *immutable*; operations that act on instances produce new instances; the process is the same as with the String class in Java.
- In this lesson we focus on LocalDate to handle date needs. It is the simplest class to use in the new API; the other classes are more complex variants of this one.
- Eventually, all date and time handling requirements in new Java projects will use the Java 8 Date and Time API. Do practice with this APIs.

See Demo : lesson3.dateapi \AgeCalculator.java and LocaleDateDemo1.java

LocalDate Sample Code

```
System.out.println("Today's date: " + LocalDate.now()); // Current date
```

```
System.out.println("Specified date: " + LocalDate.of(2000, 1, 1)); // Set a new Date
```

////////// Formatting LocalDate as strings and reading date strings as LocalDate

```
public static final String DATE_PATTERN = "MM/dd/yyyy";
```

```
public static LocalDate localDateForString(String date)      {
```

```
    return LocalDate.parse(date,  
    DateTimeFormatter.ofPattern(DATE_PATTERN));  
}
```

```
public static String localDateAsString(LocalDate date) {
```

```
    return date.format (DateTimeFormatter.ofPattern(DATE_PATTERN));  
}
```

Using this Keyword

Implicit and explicit parameters:

- Example: `e.raiseSalary(5)` actually passes in two parameters, one explicit – the number 5 -- and one implicit, which is the object reference `e`.
- Once an object has been created, methods have access to this implicit parameter through the use of the "this" keyword.

The most common reason for using the this keyword is because a field is shadowed by a method or constructor parameter.

- It is used to refer to current object
- It is always a reference to the object on which method was invoked.
- It can be used to invoke current class constructor
- It can be passed as an argument to another method

Example: You can write the constructor of Employee like this:

```
String name;
```

```
Double salary;
```

```
Employee(String name, double salary) {
```

```
    this.name = name;
```

```
    this.salary = salary;
```

```
}
```

Accessors (getters) and Mutators (setters)

- Important examples of *public methods*.
- They play the role of *getting* values stored in instance variables, and *setting* values in instance variables, respectively.
- Click Source -> Generate Getters and Setters

Example:

```
//from Advanced Employee Example
public String getNickName() {
    return nickName;
}
public void setNickName(String aNickName) {
    nickName = aNickName;
}
```


continued

- Getters and setters support "encapsulation". Permits the class that owns the data to control access to the data.

Example: Notice "name" has been made "read-only" since there is no setter, whereas "salary" is modifiable.

Sample benefit: Ability to change the implementation without undermining client code:

```
private String firstName;  
private String lastName;  
public String getName() {  
    return firstName + " " + lastName;  
}
```

Security: Careless Use of Getters

Example from Employee

```
// Getter in Advanced Employee example  
public Date getHireDay() {  
    return hireDay;  
}
```

```
//Rogue programmer could do this: to modify the date  
Employee harry = . . . //get instance  
Date d = harry.getHireDay();  
long tenYearsInMilliseconds =  
    10 * 365 * 24 * 60 * 60 * 1000L;  
long time = d.getTime();  
d.setTime(time - tenYearsInMilliseconds);
```

Question: How can this be prevented?

A Solution: Use `clone()` to correct `getHireDay()`:

```
//corrected code
public Date getHireDay() {
    return (Date)hiredDay.clone();
}
```

Moral: Do not return *mutable data fields* directly, via getter methods. Instead, return a copy of such fields.

Another Solution: Immutable Classes

- **Solution**. Use `LocalDate` in place of `GregorianCalendar` and `Date`: The `hireDay` should now have type `LocalDate`, and the year, month, day passed into the constructor should be used to construct this `LocalDate`. Then `getHireDay()` will return an immutable `LocalDate`.

How to Make a Class Immutable

- A class is immutable if the data it stores cannot be modified once it is initialized.
- Java's `String` and number classes (such as `Integer`, `Double`, `BigInteger`), as well as `LocalDate`, are immutable. Immutable classes provide good building blocks for creating more complex objects.
- Immutable classes tend to be smaller and focused (building blocks for more complex behavior). If many instances are needed, a “mutable companion” should also be created (for example, the mutable companion for `String` is `StringBuilder`) to handle the multiplicity without hindering performance.
- Guidelines for creating an immutable class (from Bloch, *Effective Java*, 2nd ed.)
 - **All fields should be *private* and *final*.** This keeps internals private and prevents data from changing once the object is created.
 - **Provide *getters* but no *setters* for all fields.** Not providing setters is essential for making the class immutable.
 - **Make the class *final*.** (This prevents users of the class from accessing the internals of the class by inheritance – to be discussed in Lesson 4.)
 - **Make sure that getters do not return mutable objects.**
- See demo `lesson3.immutable.Contacts.java`.

Immutable Classes

Mutable

```
public class IntHolder {  
    private int value;  
    public IntHolder(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return value;  
    }  
    public String SetValue(int value) {  
        this.value = value;  
    }  
}  
// main method  
IntHolder holder = new IntHolder(10);  
int v = holder.getValue(); // return 10  
Holder.SetValue(20);  
v = holder.getValue(); // return 20
```

See Demo : ImmutableDemo.java

Immutable

```
final public class IntHolder{  
    private final int value;  
    public IntHolder(int value) {  
        this.value = value;  
    }  
    public int getValue() {  
        return value;  
    }  
}  
// main method  
IntHolder holder = new IntHolder(10);  
int v = holder.getValue(); // return 10
```

Java Records

Java 16 introduced a convenient way to create immutable classes---Java *records*.

- A record is a Java class that:
 - Is final (not accessible by inheritance – see lesson 5)
 - Lists all its instance variables in its declaration
 - Provides an implicit public constructor that sets all values of its instance variables (which are private and final)
 - Provides implicit getter (accessor) methods in a compact style: For instance, if there is an instance variable `name`, the getter is `name()`.
 - Does not allow modification of the constructor, instance variables, or getter methods, and does not allow introduction of additional instance variables
 - Does allow adding instance methods and also static fields and methods.
 - Allows use of a *compact constructor* for purposes of validating parameters, which is executed before the implicit public constructor. (You cannot read or modify parameters within a compact constructor.)

The next slide shows an example of an immutable class coded in the traditional way (class `Contacts`) and also as a Java record (record `ContactsRecord`). See the demo code in `lesson3.immutable` package.


```

public final class Contacts {

    private final String name;
    private final String mobile;

    public Contacts(String name, String mobile) {
        //validate parameters
        if(name == null || name.length() == 0) {
            throw new IllegalArgumentException("Invalid input");
        }
        this.name = name;
        this.mobile = mobile;
    }

    public String getName(){
        return name;
    }

    public String getMobile(){
        return mobile;
    }

    public String toString() {
        return name + ": " + mobile;
    }

    public static void main(String[] args) {
        Contacts rec1 = new Contacts("John", "641-472-6666");
        Contacts rec2 = new Contacts("Anne", "641-919-2222");
        System.out.println("Name in record #1: " + rec1.getName());
        System.out.println("Record #1: "+rec1+"\nRecord #2: "+rec2);
    }
}

```

```

public record ContactsRecord(String name, String mobile) {
    public ContactsRecord {
        if(name == null || name.length() == 0) {
            throw new IllegalArgumentException("Invalid input");
        }
    }

    public String toString() {
        return name + ": " + mobile;
    }

    public static void main(String[] args) {
        ContactsRecord rec1 = new ContactsRecord("John", "641-472-6666");
        ContactsRecord rec2 = new ContactsRecord("Anne", "641-919-2222");
        System.out.println("Name in record #1: " + rec1.name());
        System.out.println("Record #1: "+rec1+"\nRecord #2: "+rec2);
    }
}

```

In ContactsRecord:

- name and mobile are declared with the record
- name and mobile are private final instance variables with accessors name() and mobile()
- Values for name and mobile are set by the implicit constructor (see main method)
- ContactsRecord is a compact constructor that validates parameters (it does not and cannot set values for instance variables)
- This record provides its own toString method; the code here overrides it to make it useful



Day - 2

Outline of Topics

- The Object-Oriented Paradigm and OO Concepts
- Template for User-Defined Classes in Java
 - Creating new objects
 - Accessing data and operations in an object
 - Access Modifiers
 - Destroying objects
- Some Classes in the Java Library: `Date`, `GregorianCalendar`, `LocalDate`
- The `'this'` keyword
- Mutators (setters) and Accessors (getters)
- How to Make a Class Immutable
- Boxed Primitives and Autoboxing
- Enum Constants
- Static Fields And Methods
- How Objects and Variables Are Stored in Memory
- Call by Reference vs Call by Value
- Import Packages
- Principles of Good Class Design

Final Instance Fields = Constants

- When the final keyword is used for an instance variable, it means the variable may not be used to store a different value.
- Since the name field in Employee can never change, we could make it final
- `private final String name;`
- Final instance variables often represent constants. Recall `Math.PI`

See Demo : `FinalExample.java`

```
class LabelConstants {  
    public final static int LEFT = 0;  
    public final static int CENTER = 1;  
    public final static int RIGHT = 2;  
}
```

- One problem with the above approach can be seen in the following code:

```
Label lab = new Label("hello");  
lab.setAlignment(5);
```

There is no compiler-based control over the use of “alignment” constants for Labels.

A more reliable way to store constants is to use an *enumerated type* (also called an *enumeration type*). An enumerated type is a class all of whose possible instances are explicitly enumerated during initialization.

Enumerated type

- create an enumerated data type in a statement that uses the keyword **enum**, an identifier for the type, and a pair of curly braces that contain a list of the enum constants, which are the allowed values for the type.
- Example:

```
enum LabelConstant{ LEFT, CENTER, RIGHT };
```

```
enum Size { SMALL, MEDIUM, LARGE};
```

```
//usage: if(requestedSize==Size.SMALL) applyDiscount();
```

```

enum Currency {
    US, INDIA, UK
}
class sample1 {
    void display(Currency cy) {
        switch (cy) {
            case US -> System.out.println("Dollar");
            case INDIA -> System.out.println("Rupee");
            case UK -> System.out.println("Pounds");
        }
    }
}

```

See Demo : EnumDemo.java, EnumDemo1.java

```

public class EnumDemo {
    public static void main(String[] args){
        sample1 s = new sample1();
        s.display(Currency.UK);
        s.display(Currency.INDIA);
        s.display(Currency.US);
        // Convert String to Enum
        String x = "uk";
        x = x.toUpperCase();
        Currency input = Currency.valueOf(x);
        s.display(input);
    }
}

```

Boxed Primitives

- Java provides **immutable wrapper classes** for all primitive types:

<code>int -> Integer</code>	<code>char -> Character</code>
<code>float -> Float</code>	<code>short -> Short</code>
<code>double -> Double</code>	<code>byte -> Byte</code>
<code>boolean -> Boolean</code>	

- Conversions between primitives and wrapper classes make use of methods with the same names. For example:

Given `Integer x,`

```
int u = x.intValue();
```

Given `int y,`

```
Integer v = Integer.valueOf(y)
```

Given `Double x`

```
double u  
    = x.doubleValue();
```

Given `double y,`

```
Double v = Double.valueOf(y)
```

Refer: `lesson3.localfieldinit\BoxedPrimitive`

(continued)

- *Autoboxing*. Usually, it is not necessary to explicitly perform a conversion – the compiler will take care of the conversion automatically. For example:

```
Integer[] arr = { 2,3,4,5};  
int x = 10;  
arr[0] = x; // auto boxing int=> Integer
```

- *May Be Null*. Unlike their primitive counterparts, wrapper classes may be null. If so, exceptions may be thrown. For example:

```
Integer x;  
System.out.println(x.intValue()); //NullPointerException
```

Boxed Primitive Methods

- Each of the wrapper classes has a few methods (we have seen `valueOf` and `intValue` for `Integer`)
- `Integer.parseInt` (likewise, `Double.parseDouble`, etc.)
Example: `String aNumber = "45";`
`Integer num = Integer.parseInt(aNumber);`
- `compareTo` - works the same way as `compareTo` for Strings

Example:

```
Integer x = 3;
Integer y = 4;
if(x.compareTo(y) < 0)
    System.out.println("x is smaller than y");
else if (x.compareTo(y) == 0)
    System.out.println("x and y are equal");
else { //x.compareTo(y) > 0
    System.out.println("x is greater than y");
}
```

Static Fields

- *Static fields* – uses the keyword **static** as part of the declaration
- If a field is static, there is only one copy for the entire class, rather than one copy for each instance of the class.
- Like a global variable for that class, it is visible to all objects of the class
- A value for a static variable is the same for all instances of the class

Example

"Create a Java class that keeps track of how many instances of itself have been created. This data should be stored in a variable and should be accessible by a public accessor method.

Write a main method that constructs several instances of the class and then outputs the number of instances created by calling this accessor method."

Static Fields

```
public class Prob4 {  
    static int count=0;  
    Prob4(){ ++count;}  
    public int getCount(){return count;}  
  
    public static void main(String[] args){  
        Prob4 instance = null;  
        for(int i = 0; i < 10; ++i){  
            instance = new Prob4();  
        }  
        System.out.println("Number of instances so far  
                             "+instance.getCount());  
    }  
}
```

Static Methods

- A Method can also be declared as Static
- Typically these are utility methods that provide a service of some kind, like a computation.
- Can be accessed without an instance of enclosing class
- Cannot access instance variables
- Does not have an implicit parameter (so, cannot be used with "this")
- Good example is main() method. It is called before any object of the class is created.

Typical form of static method is

`public static <return_val_type> method(params)`

How to make static method calls:

By convention, always use `<class_name>.<method_name>`

Example: `Math.pow(2,5);`

Eg: `StaticExample.java`, `StaticExample1.java`

Static Methods

```
public class Prob4 {  
    static int count=0;  
    Prob4(){ ++count;}  
    public static int getCount(){return count;}  
  
    public static void main(String[] args){  
        Prob4 instance = null;  
        for(int i = 0; i < 10; ++i){  
            instance = new Prob4();  
        }  
        //Can use any instance. Really only have access to last one!  
        System.out.println("Number of instances so far =" +instance.getCount());  
        System.out.println("Number of instances so far =" +Prob4.getCount());  
    }  
}
```

How Objects and Variables Are Stored in Memory

- **Registers** – fastest area of memory (within the processor) but no access with Java programs.
- **Stack Memory** - second fastest area of RAM because of direct support from processor for the stack pointer. Stack pointer is moved down to access new memory and back up to release it. **Primitive Types and Address of objects, method calls and local variables.**
- **Heap Memory** –A general-purpose area of RAM where all Java objects are stored. **The instance fields (non-static variables) of objects also reside in the Heap.**
The String Pool. The String class maintains in heap memory a table of Strings that have been "interned". When a **string literal is defined**, the table is checked; if the string already exists, it is returned, otherwise it is added to the table. Two interned strings that have the same values will always be considered equal using == since they are literally the same object.
- https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf

How Objects and Variables Are Stored in Memory

- **Static Storage**

- This is a memory area where static variables and static object references are stored.
 - Static variables and references are associated with the class rather than instances of the class.
 - These references remain "alive" through the entire execution of the program.
- Prior to JDK 8, static variables were stored in the PermGen space.
- Starting with JDK 8, the storage for class metadata, including **static fields and static methods, are in the Meta space**.
- Class metadata consists of information about the structure and behavior of a class need to be loaded at runtime.
- Metaspace is a crucial component of the JVM's memory management strategy, designed to handle class metadata efficiently by using native memory rather than the heap.
- **Native memory** refers to the memory that is managed by the operating system, outside the control of the Java Virtual Machine (JVM).

Example

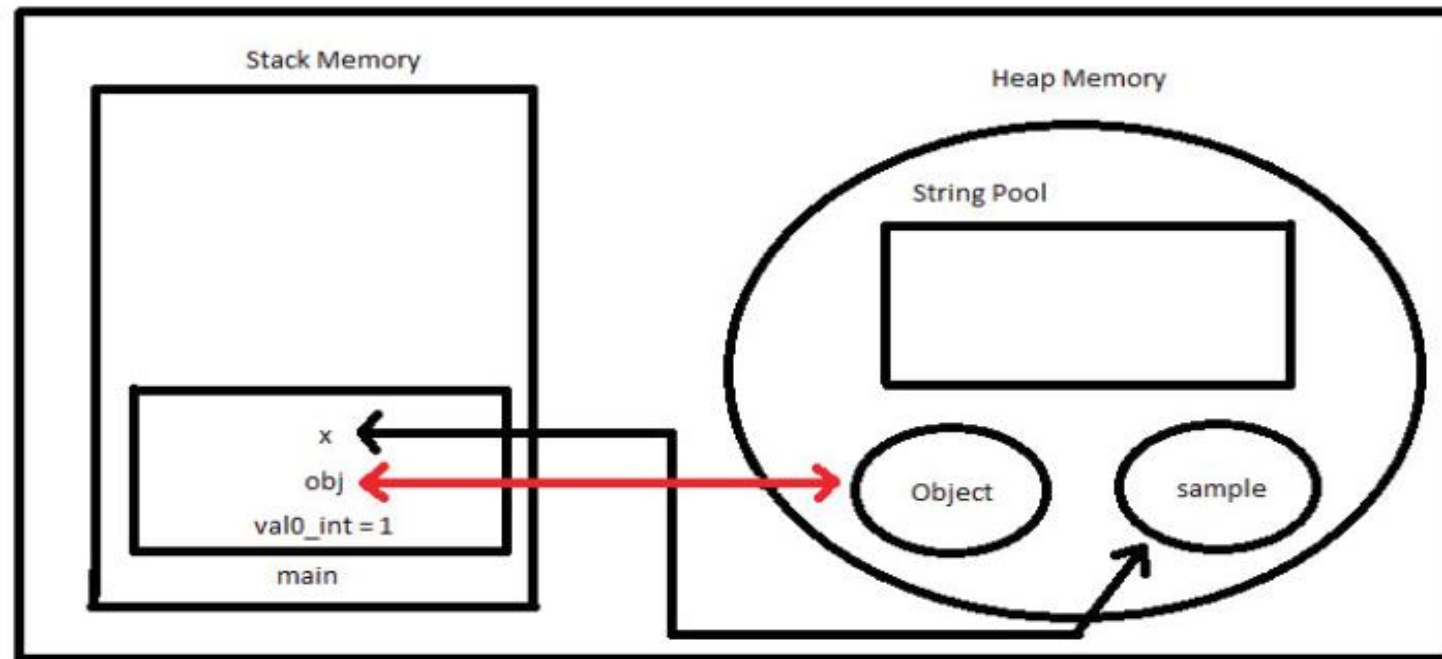
```
class Sample {  
    Sample x;  
    public static void main(..) {  
        x = new Sample();  
        Object obj = new Object();  
        int val0_int = 1;  
    }  
}
```

Stack Memory

<i>vble name</i>	<i>address</i>
val0_int	1
x	@1f77ac92
obj	@1f89ab83

Heap Memory

<i>address</i>	<i>object</i>
@1f77ac92	Sample
@1f89ab83	Object



See video: <https://www.youtube.com/watch?v=45omaTzSlvA&t=27s>

Main Point 2

Static fields and methods are fields and methods whose lifetime persists throughout execution of the application, and when used with the public keyword, are globally accessible. The notion of "static" parallels the recognition that there is a field in life that is globally available and is always located in the same place in “memory”: namely, pure consciousness.

Miscellaneous: **Overloading**

The signature of a method is the combination of the method's name along with the number and types of the parameters (and their order).

You can overload as long as signatures are different. It applies to methods and constructors.

Overloading

- **Overloading a constructor**

Example: Recall two of the constructors from `GregorianCalendar`

- `GregorianCalendar()`
- `GregorianCalendar(int year, int month, int day)`

- **Overloading methods**

- Methods can be overloaded using the same rules.

`int myMethod(int input1, String input2)`

`void myMethod(double input1, String input2)`

- One constructor can call another constructor using this keyword.
- Calling constructor should be the first statement.
- Constructor cannot call itself because it will result in a recursive call.

Demo : `OverLoadDemo.java` , `AreaConstructor.java`

Miscellaneous: Field Initialization

- *Explicit field initialization.* When a class is constructed, all instance variables are initialized in the way you have specified, or if initialization statements have not been given, they are given default initialization.
 - primitive numeric type variables (including `char`) -- default value is 0;
 - `boolean` type variable – default value is `false`
 - object type variables (including `String` and array types) – default value is `null`
- It is not necessary to initialize *instance* variables in your code – you can use the default assignments. However, it is good practice to initialize always.
- By contrast, local variables (variables declared within a method body) *should always* be initialized (typically compiler error otherwise). Local variables are *never provided with default values*.

```
int x; // Default - 0
float f; // Default - 0.0
char c; // Default - '\u0000' -
Empty
String s; // Default - null
double d; // Default - 0.0
boolean b; // // Default - false
```

Order of Execution in a Class

When a class is used for the first time, it needs to be loaded.

1. After a class is loaded to the memory, its static data fields and static initialization block are executed in the order they appear in the class. (Static fields are initialized only once; static blocks executed only once.)
2. Instance initialization block : It is initialized immediately after all static initialization has occurred (and before any instance variables are initialized).

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
 2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after `super()` constructor call).
 3. The instance initializer block comes in the order in which they appear.
3. All instance variables are initialized with their default values
 4. If the first line of the constructor calls another constructor, the body of the second constructor is executed, then the body of the first constructor is executed.

See Demo : `lesson3.orderofexec \ClassE.java`

Private constructors and singleton classes

Occasionally, constructors are declared to be private

- A private constructor cannot be accessed by any other class. The only way for another class to communicate with such a class is by way of *static methods*.
- Useful for utility classes which provide static methods only – example: Math (this class has a private constructor).
- The Singleton's purpose is to control object creation, limiting the number of objects to only one.
- Private constructor cannot be accessed by any other class.
- Private constructor can be used to ensure that *only one instance* of a class is ever used. (Such a class is called a *singleton*. Making a class as singleton during design is called *the Singleton design pattern*.)
- Singletons often control access to resources.

Packages in Java

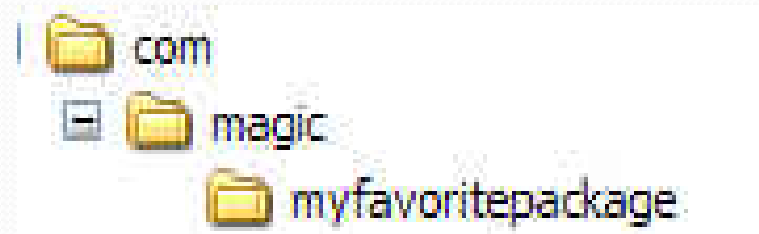
1. Packages represent units of organization of the classes in an application. The basic rules for packaging Java classes are like the rules in a file system: A package may contain Java classes and other packages (and other resources, like image files).
2. It is not *necessary* to use packages in a Java application, and for small applications, there is often no need to use them. In that case, the JVM creates a *default package* and views your classes as belonging to this package.

3. For medium to large applications, it is important to organize code in packages for several reasons:

- Packaging according to a systematic scheme (so that classes that are related to each other belong to the same package and unrelated classes belong to different packages) supports parallel development of code and makes code easier to maintain.
- Packages can be defined so that they can be reused by other applications.
- A package creates a *namespace* that helps to prevent naming conflicts. For example, it is possible to have two classes with the same name as long as they belong to different packages.

4. Conventions concerning packages

- The name of a package should consist of *all lower case letters*.
- Example: `myfavoritepackage` //correct
- `myFavoritePackage` //incorrect
- To avoid naming conflicts between packages developed in different places (even possibly different parts of the world), a package “nesting” convention has developed in the Java community: Name your package by using your company’s domain name in reverse as the prefix.
- Example: Your company’s domain name is `magic.com`. Your top-level package is `myfavoritepackage`. So, for production, name this package
- `com.magic.myfavoritepackage`



Importing Classes

Can use fully qualified class names or imports or both

```
package mypackage;  
import java.lang.Math;  
MyClass {  
    public static void main(String[] args) {  
        Math.sqrt(4);  
    }  
}
```

OR

```
package mypackage;  
  
MyClass {  
    public static void main(String[] args) {  
        java.lang.Math.sqrt(4);  
    }  
}
```

Static Imports

- Static imports (new with j2se5.0) allows you to import static fields and methods.

Example:

```
package mypackage;  
import static java.lang.Math.*;  
MyClass {  
    public static void main(String[] args) {  
        sqrt(4);  
        System.out.println(PI);  
    }  
}
```

Caution: Use of static imports is often considered a bad practice because of readability – it is too hard to determine the origin of a method that has been statically imported.

Call by Reference vs Call by Value

- A programming language supports a "call by reference" idiom for method calls if the values stored in the variables that are passed into the method may be modified in the method body.

A programming language supports a "call by value" idiom for method calls if the values stored in the variables that are passed into the method represent *copies* of the original values, so that they may *not* be modified in the method body. ***Java uses call by value.***

- Call By Value for Primitives

```
public static void main(String[] args) {  
    CallByValuePrimitives c = new CallByValuePrimitives();  
    int num = 50;  
    c.triple(num);  
    //value of num is still 50  
    System.out.println(num);  
}  
  
public void triple(int x) {  
    x = 3 * x;  
}
```

- Change Value Stored in an Object Reference

```
public static void main(String[] arg) {  
    ChangeValueInReference c = new ChangeValueInReference();  
    Employee harry = new Employee("Harry", "Rogers", 50000, 1989, 10, 1);  
    c.tripleSalary(harry);  
    //salary has been tripled  
    System.out.println("Harry's salary now: " + harry.getSalary());  
}  
  
public void tripleSalary(Employee e) {  
    e.raiseSalary(200);  
}
```

- Call by Value for Objects

```
public class CallByValueObjects {  
    public static void main(String[] args) {  
        CallByValueObjects c = new CallByValueObjects();  
        Employee a = new Employee("Alice", "Thompson", 60000, 1995, 2, 10);  
        Employee b = new Employee("Bob", "Rogers", 70000, 1997, 10, 1);  
        c.swap(a, b);  
        //To which Employee does the reference a point?  
    }  
    public void swap(Employee x, Employee y) {  
        Employee temp = x;  
        x = y;  
        y = temp;  
    }  
}
```

Moral: all method calls in Java are **call by value**.

Demo Code : lesson3. callbyvaluereference

**Ref : For more information read Pages 233-244. Beginning Java Fundamentals by Kishori Sharan from Apress.
Read this thru Sakai Resources → Textbooks.**

Principles of Good Class Design

1. Keep data private and represent the services provided by a class with public methods
2. Always initialize data
3. Divide big classes into smaller classes (if too many fields or too many responsibilities)
4. Not all fields need their own accessor and mutator methods; use this flexibility to control access to fields – e.g. can make a field read-only by providing a getter but no setter.
5. Use a consistent style for organizing class elements within each class
6. Follow naming conventions for packages, classes, and methods
7. **Ambler's Book.** Scott Ambler has systematized many best coding practices and an optimal coding style in his book *The Elements of Java Style*.

Refer : Additional Reading part in Sakai (Ambler's book)

Main Point 3

Java method calls are in every case *call by value* (and never *call by reference*). Even though an object reference can be passed into a method, the variable that stores the reference cannot be made to point to a different reference within the method. Therefore, only a *copy* of such a variable is ever passed to a method (in other words, call by value). Call by value is reminiscent of the incorruptible quality of pure consciousness – "fire cannot burn it, nor water wet it".

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Object identity and identifying with unboundedness

1. A Java class specifies the type of data and the implementation of the methods that any of its instances will have.
- Every object has not only state and behavior, but also identity, so that two objects of the same type and having the same state can be distinguished.

-
1. **Transcendental Consciousness** *is the identity of each individual, located at the source of thought.*
 2. ***Impulses within the Transcendental field:*** *When consciousness knows itself it creates the lively impulses of pure knowledge within the field of silent Being.*
 3. **Wholeness moving within itself:** *In Unity Consciousness, one's unbounded identity is recognized to be the final truth about every object. All objects are seen to have the same ultimate identity, even though differences on the surface still remain.*

