Classes and Objects

Classes and Objects

CLASSES AND OBJECTS - RECAP

- Class
  - Template used to create objects
  - Defines State and Behavior of an object
  - State – Instance variables declared within a class
  - Behavior –Instance methods or functions declared in a class
  - A class encapsulates the data and functionality

- Objects
  - instances created using a class
  - Class is said to be instantiated when an object is created
  - Each object has its own state stored in member or instance variables
  - Exposes its behavior through methods
  - Has unique identity

Components of a class

COMPONENTS OF A CLASS

Class Name

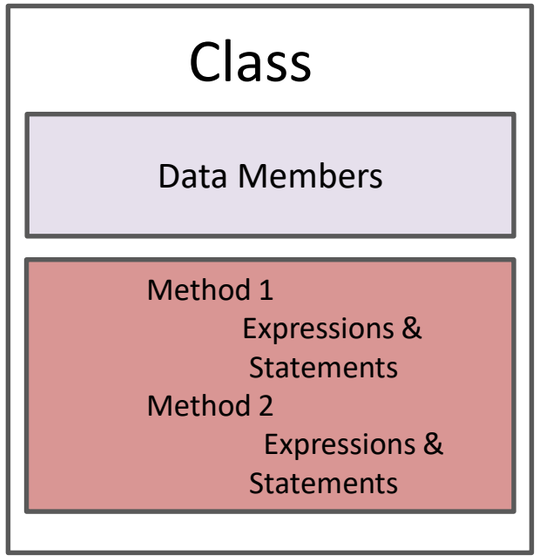
Static Variable

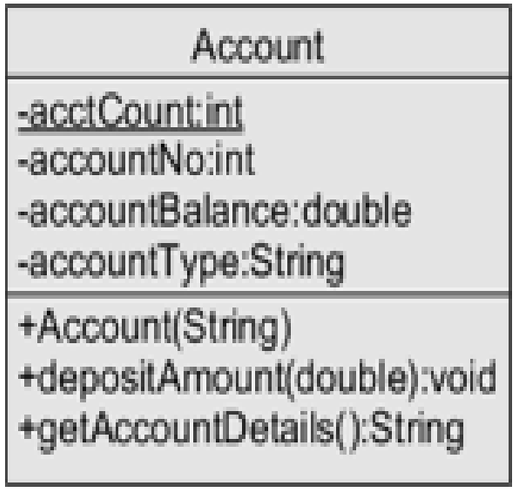
Instance Variables

Constructor

Methods

```
public class Account {
    static int acctCount = 0;

    int accountNo;
    double accountBalance;
    String accountType;

    public Account(String acType) {
        accountNo = ++ Account.acctCount;
        accountBalance = 0;
        accountType = acType;
    }

    public void depositAmount(double amount) {
        accountBalance += amount;
    }

    public String getAccountDetails() {
        String format = "Acct No : "+accountNo
         +"\n" + "Acct Balance: "+accountBalance;
        return format;
    }
}
```





INSTANCE VARIABLE

- Stores the state of objects
- Declared outside methods and statement blocks in a class
- Can be directly accessed in member methods/blocks within a class

Syntax

```
<modifier> <type> <identifier> [= <initial value>];
```

```
private int accountNo;
private String customerName;
```

Note

- Instance variables are initialized to default values of their data types during object creation
- Every object has its own copy of instance variables in memory

STATIC VARIABLE

- Used to store common property shared by all objects of a class
- Only one copy of static variable is created in memory
- Belongs to the class and not an object

```
private static int acctCount;
private static final double TAX_RATE = 9.0;
```

Note

- Static variables are created and initialized only once when the class is first loaded

CONSTRUCTOR

- Special method used for initialization of object
  - i.e. giving initial values to instance variables
- Has the same name as the class
- Does not return any values
- gets invoked automatically when an object is created

```
//NO ARGUMENT CONSTRUCTOR

public class Account{
   public Account() {
            accountNo = ++acctCount;
            accountBalance = 0;
   }
}
```

PARAMETERIZED CONSTRUCTOR

- During object creation, values can be assigned to instance variables, by passing parameters to the constructor

```
//PARAMETERIZED CONSTRUCTOR
public Account(String accType,double
accBalance) {
        accountNo = ++acctCount;
        accountBalance = accBalance;
        accountType = accType;
}
```

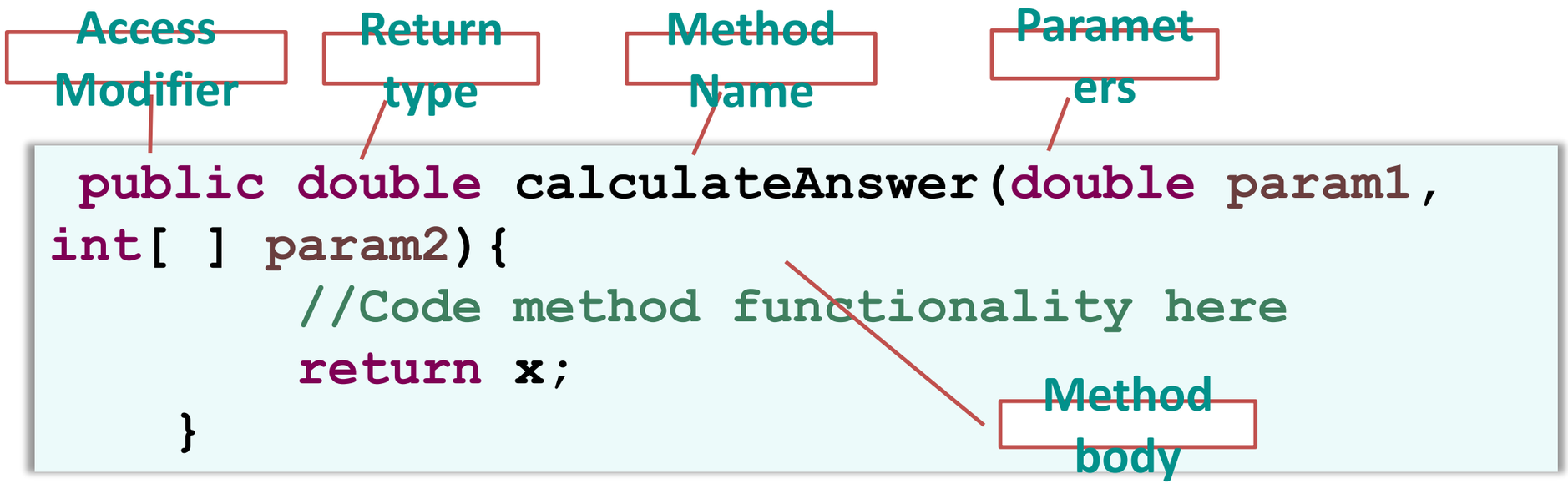
★ **Note**

- uctor is provided by programmer

```
public Account() { super();
}
```

- If programmer codes a constructor, Java doesn't provide default constructor

METHOD DECLARATION



- Method represents the behavior of objects
  - Are identified by a method name which is unique in a class
  - Used to define object functionalities
  - Accept input parameters and return result

- Method needs to be invoked using a object reference, to execute their functionality

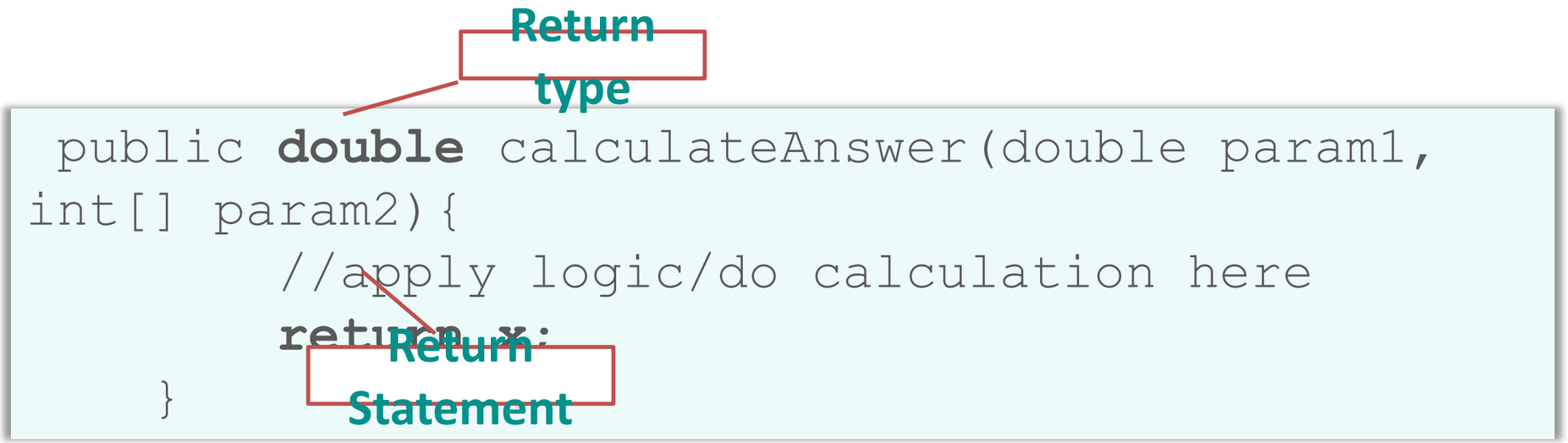
METHOD PARAMETERS


```
 public double calculateAnswer(double param1,
int[] param2){
        //apply logic/do calculation here
        return x;
    }
```

- Are the inputs provided to the method during execution
- Declared in parentheses after method name
- Every parameter has a datatype and a name

- Method can take zero or multiple parameters
- Parameters can be of **primitive or reference** data type
- Parameter values are passed, when the method is called/invoked

METHOD RETURN TYPE



- Methods can return back values as output using **return** statement
- During method declaration, return type defines the datatype of variable the method returns
- Return type can be primitive data type or reference data type
- Methods not returning anything should declare return type as “void”
- Any value that can be implicitly converted to the return type can be returned

METHOD BODY

```
 public double calculateAnswer(double param1,
int[] param2){
        //apply logic/do calculation here
        return x;
    }
```

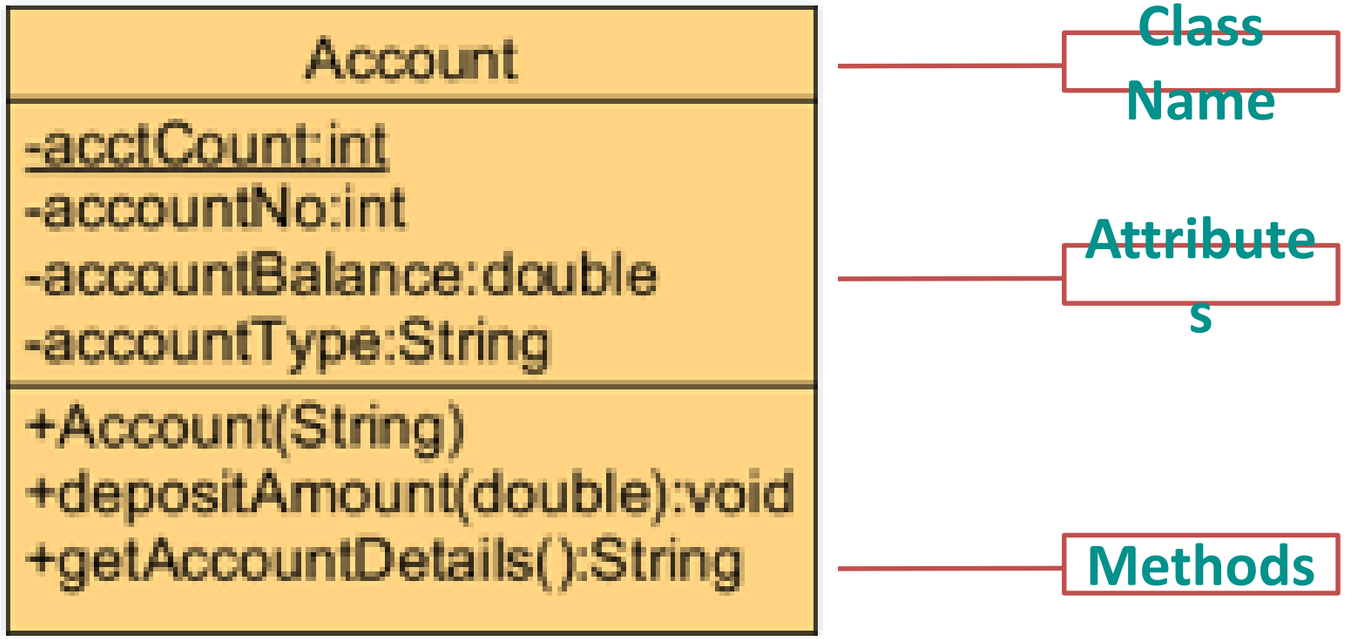
- Method Body is written within curly braces after parameter declaration
- Functionality of the method i.e. business logic/calculation is coded here
- Method can access instance variables in the body
- One method can directly invoke another method of the same class without explicit reference

LOCAL VARIABLE

- Variables declared within the method body
- Cannot be accessed outside the method
- Have to be compulsorily initialized before using, to avoid compilation failure

```
public String getAccountDetails() {
    String format = “”;
    format = "Acct No : “ + accountNo;
    return format;
}
```

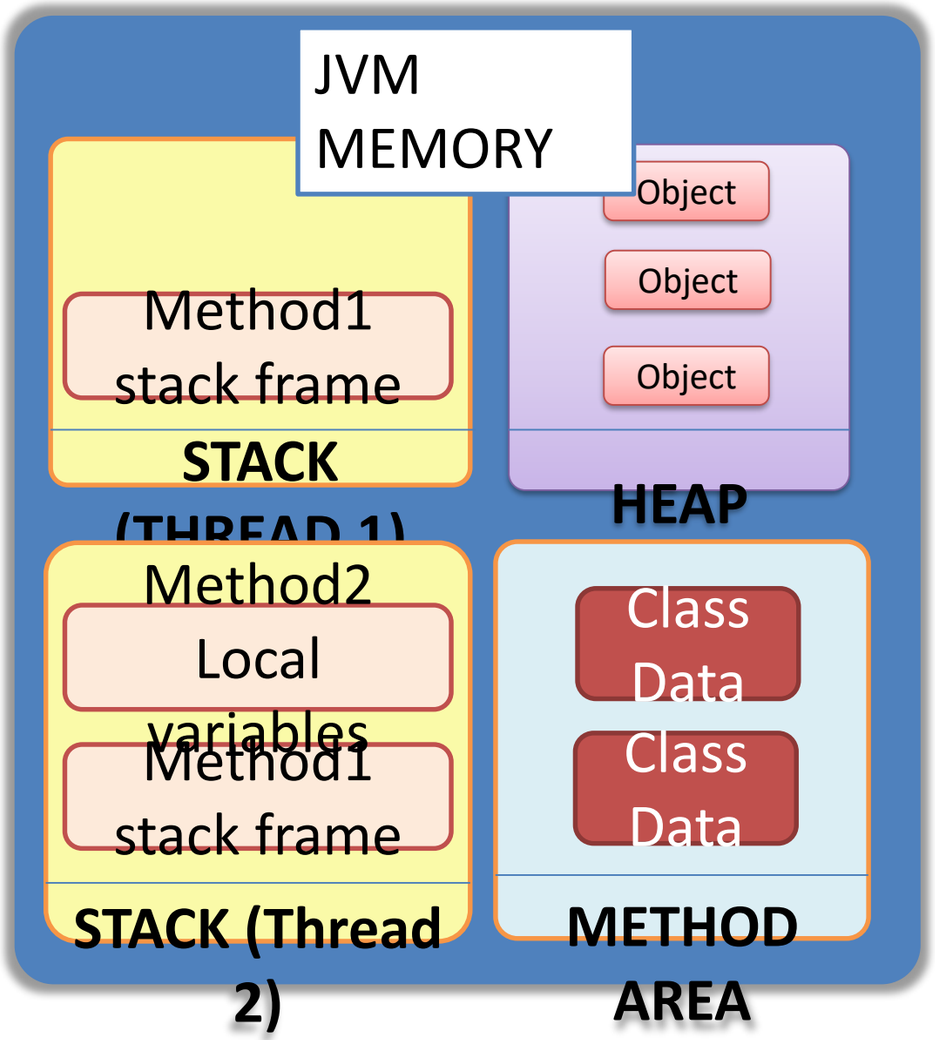
CLASS REPRESENTATION USING UML



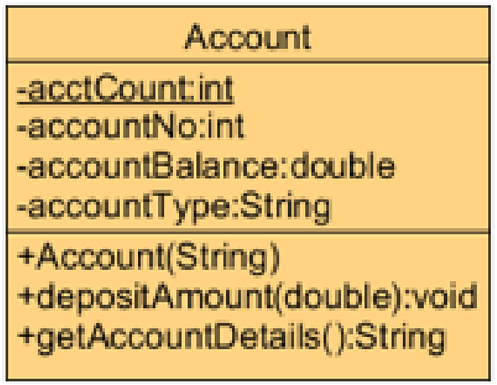
Object creation

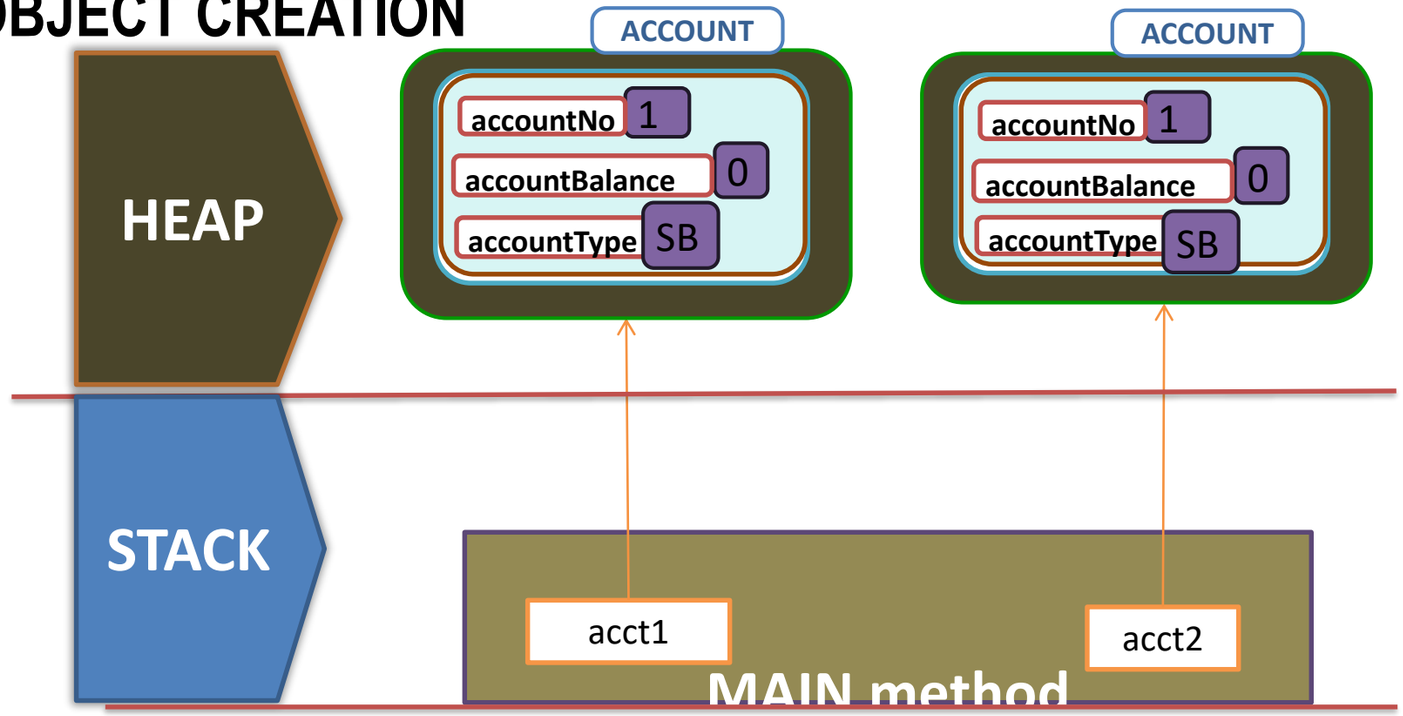
JVM Memory Model

- JVM organizes the memory it needs to execute a program into several runtime data areas
  - Heap
    - Memory area where objects are created
    - Shared by all threads
  - Stack
    - Stores the state of Java method invocations for a thread
    - Every thread has a separate stack
  - Method Area
    - Contains data about the class parsed from the class file



OBJECT CREATION






```
class TestAccount {
public static void main(String[]
args) {
   Account acct1 =
                  new Account();
   Account acct2 = new Account();
}
```

Custom Datatype

Reference variable

new operator creates an object on Heap and returns its address

ACCESSING DATA MEMBERS

- Reference variable is used to access objects instance variable's and to invoke methods
- Reference variable should be of same type as the Class of the object

```
Account acct1 = new Account();
```

- Access instance variable using dot notation

referencevariable.instancevariable

```
int accNo = acct1.accountNo;
System.out.println("Acct Type  " +
acct1.accountType);
```

★

Note

- instance variables can be directly accessed within the methods of the same class without explicit reference

INVOKING METHODS

- Methods are invoked using reference variable with a dot notation

```
referencevariable.method()
```

-

```
Account acct1 = new Account();
double depAmt = 1000;
acct1.depositAmount(depAmt);
```

-

```
String details =
acct1.getAccountDetails();
```

```
public class Account {
    int accountNo;
    double accountBalance;

    public void depositAmount(double
amount) {
        accountBalance += amount;
    }

    public String getAccountDetails() {
        String format =              “Acct No : "+
accountNo  +"\n"
                + "Acct Balance:
```

INVOKING METHODS

```
double depAmt = 1000;
acct1.depositAmount(depAmt);
```

- When depositAmount(..) method is called, the value of depAmt (i.e. 1000) is copied in to parameter variable amount

```
public class Account {
   ...
   public void depositAmount(double
amount) {
      accountBalance += amount;
   }
   ...
```

Note

- Methods can directly invoke other methods in the same class without a explicit reference variable
- Method in classA can invoke a method of ClassB, using a reference of ClassB object

ACCESSING STATIC VARIABLES

- Static variables can be accessed using a Class Name with a dot notation

  `ClassName.staticVariable`

```
int counter = Account.accountCount;
```

- Static variables can be accessed within and outside the defined class (Non private)

- Static variables can also be accessed using a object reference, but is not recommended

```
Account acct1 = new Account();
int counter = acct1.accountCount;
```

Compiler provides a warning for the above access

STATIC METHODS

- Methods having static keyword in method declaration

```
public static String generateAccNo(){
return "A" + acctCounter++;
}
```

- Can be invoked using a Class name as shown below
- Do not have access to instance variables

```
Account.generateAccNo();
```

- Why Static methods
  - Common utility functions such as addition of numbers, sorting of arrays etc. are not dependent on state of objects
  - Static methods make it possible to invoke such functionalities without a object instance

VARIABLE SCOPES

- Instance Variables
  - Stored in Heap memory along with the object
  - Accessible as long as the object is in heap and is referenced

- Static variables
  - Stored in method area memory along with the class object
  - Accessible as long as the class is loaded in memory

- Local variable
  - Stored in stack of the thread which is executing the method
  - Accessible as long as the method in which the variable is declared is executing

‘this’ REFERENCE

- “this” keyword is used to refer to the current executing object

```
class Employee {
     String empName;

      Employee(String empName){
           empName = empName;
     }}
```

```
Employee emp = new
Employee(“John”);
System.out.println(emp.empNa
me);

Output : null
```

- In the above example, since empName local variable shadows the instance variable, the instance variable will not be assigned the value “John”

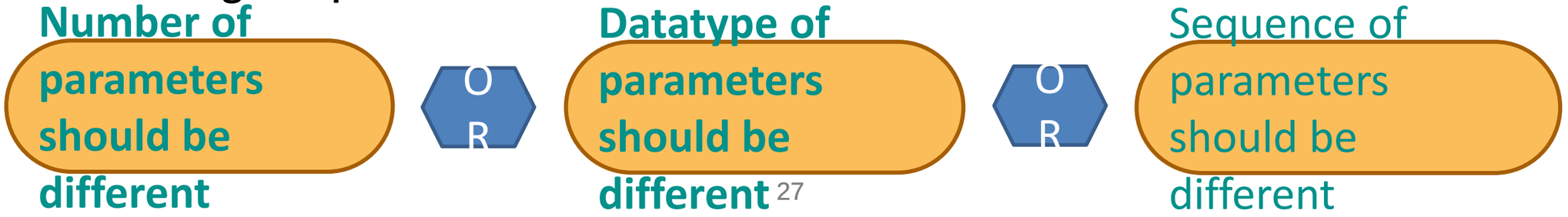
```
      Employee(String empName){

     this.empName = empName;
     }
```

- Can also be used for better readability

Overloading

METHOD OVERLOADING

- Concept of defining more than one method with same name and different parameter list
- Done for reusing method name for similar functionalities with different inputs
  - Ex: searchEmployee by Name, searchEmployee by EmpId
- Method call is linked to appropriate method during compile time based on parameters – static polymorphism
- Increases readability of the program
- Rules for Overloading
  - Method Name should be same
  - **MUST** change the parameter list in overloaded method



METHOD OVERLOADING

```
public int add (int num1,int num2) {
    return num1 + num2;
}
```

Overloading by changing datatype

```
public float add (float num1,float
num2) {
return num1 + num2;
}
```

Overloading by changing Number of parameters

```
public  int add (int num1, int num2,
int num3) {
    return num1 + num2 + num3;
}
```

CONSTRUCTOR OVERLOADING

- Like methods, Constructors can also be overloaded in a class
- Constructor overloading provides the ability to have multiple constructors in a class with different parameters

```
public class Employee {
    static int counter;
    int empId;
    String empName;

    Employee() {
      empId = ++counter;
     }
    Employee(String name) {
      empId = ++counter;
      empName = name;
    } }
```

Overloaded Constructor

```
Employee emp = new Employee();
Employee emp = new Employee("John");
```

EXPLICIT CONSTRUCTOR INVOCATION

- A constructor can explicitly invoke an overloaded constructor of the same class by using "this()"
- If present, this() should be the first statement in the constructor

```
public Account(){
    accountNo = generateAccNo();
}
public Account(double amt){
    this();
    accountBalance = amt;
}
public Account(double amt,String opDate){
    this(amt);
    accountOpenDate = opDate;
}
```

Initialization Block

INITIALIZATION BLOCK

- Initialization block is a block of statements inside curly braces

```
public class Employee {
    {
        //initialization code here
    }
}
```

- Executed when a object of the class is created
- Runs before the constructors are invoked
- Multiple initialization blocks can be present in a class and they will be executed in the sequence they are coded
- Typically used when some processing is required to initialize instance variable
- Can also be used to do common processing for all constructors in the class

STATIC INITIALIZATION BLOCK

```
public class Employee {
   static {
      //initialization code
here
   }
}
```

- Executed when the class is first loaded
- Typically used to initialize static variables
- Do not have access to instance variables
- Class can have multiple initialization block

EXECUTION SEQUENCE

```
public class Employee {
  private static int empCounter = 100;
  private int empId = 0;

  {
    System.out.println("init block executed");
  }

  static{
    System.out.println("static block executed");
  }

  public Employee(){
    System.out.println("Constructor executed");
  }

  public static void main(String[] args) {
        Employee e1 = new Employee();
  }
}
```

- Sequence of Execution
  - Class Loaded
  - Static variables initialized
  - Static block executes
  - Instance variables initialized
  - Initialization block executes
  - Constructor executes
  - Object created

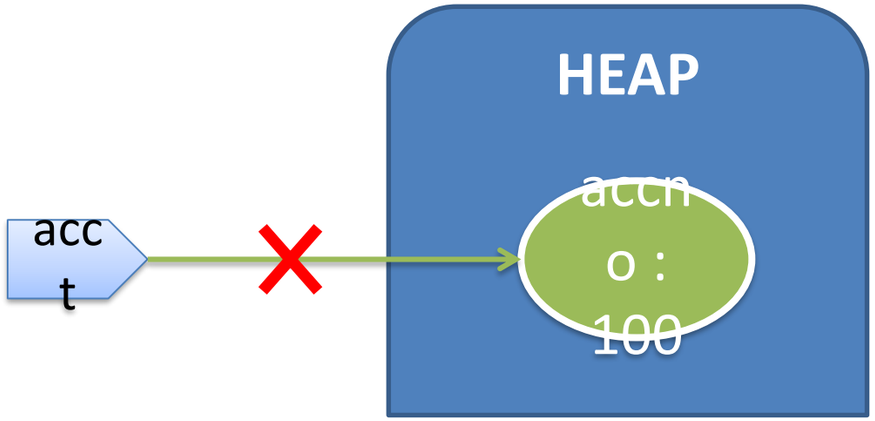
Garbage Collection

GARBAGE COLLECTION

- Garbage collection is the process of cleaning heap memory by removing unreferenced objects
- Done by a daemon thread called Garbage Collector
- Garbage collector runs automatically at frequent intervals based on memory availability
- Can be called explicitly using System.gc() but no guarantee that it will run

- When do object become garbage collectible ?

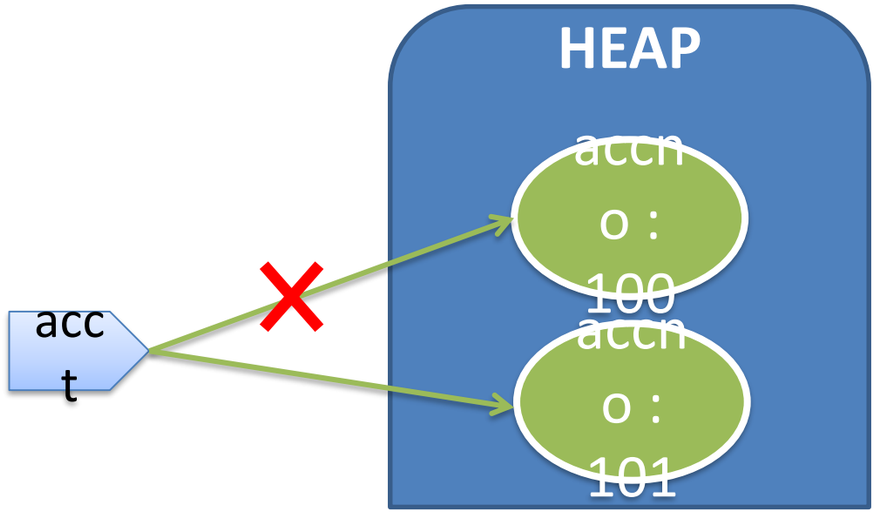
```
public class TestAccount{
  public static void
main(String[] args){
      Account acct = new
Account();
      acct = null;
```



GARBAGE COLLECTION

- When do object become garbage collectible ?
  - Reference reassigned

```
public class TestAccount{
  public static void
main(String[] args){
      Account acct = new
Account();
      acct = new Account;
}}
```



  - When objects have no reference from stack after program execution completes

Packages

PACKAGES

- A package is a collection of related classes and interfaces in Java
- A group of related classes are bundled inside a package
- Java API is grouped in to multiple packages based on the functionality
  - java.lang : contains the fundamental classes
  - java.util : contains utility classes

- Why use packages
  - Avoids naming conflict
  - Provides access protection
  - Helps easily locate classes, interfaces and other types in the huge API
  - Helps in functionality wise grouping

SPECIFYING PACKAGES

- To put a class in the package, include the package statement with the package name as shown below

```
package com.banking;
public class Account{
    //code goes here
}
```

- package should be the first statement in a class/interfaces
- Fully qualified name of the class in a package is <packagename>.<classname>
  - com.banking.Account
- Subpackages are packages created inside other packages and are treated as separate package
  - Ex. com.banking.loan is a sub-package of com.banking

IMPORTING PACKAGES

- To use a class of one package, in a class of another package, the class has to be imported using import statement

```
package
com.banking;
public class
Account{
    //code goes
here
}
```

```
package com.test;
import com.banking.Account;
public class TestAccount{
 public static void main(String[]
 args) {
    Account acct = new Account();
 }}
```

- There can be multiple import statements in a class for importing multiple classes/interfaces

SOME JAVA API PACKAGES

package name	Description
java.lang	**Contains** classes that are fundamental to the design of the Java programming language
java.util	**Contains utility classes like Calendar, Scanner, Date, Collection classes etc**
java.io	**Contains classes for** system input and output through data streams, serialization and the file system
java.text	Contains classes and interfaces for handling text, dates, numbers, and messages
java.awt	**Contains classes for creating GUI and for painting graphics and images**
java.sql	**Contains API for accessing and processing data stored in a data source (usually a relational database)**

Access Control

ACCESS MODIFIERS

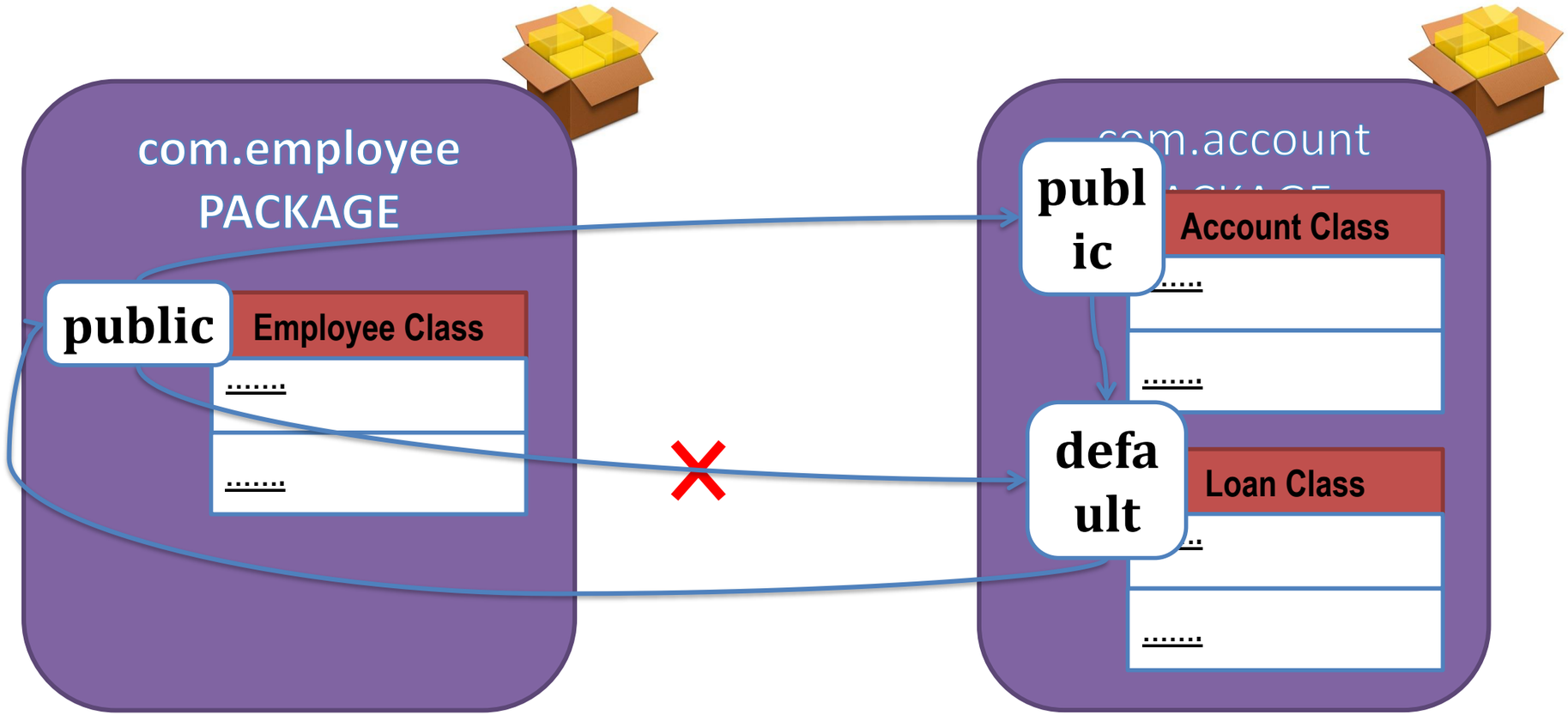
- Used to expose or hide the attribute and behaviour of a class
- provide different access levels to the class/types and its members

- public (+)
  - declaration that is accessible to all classes

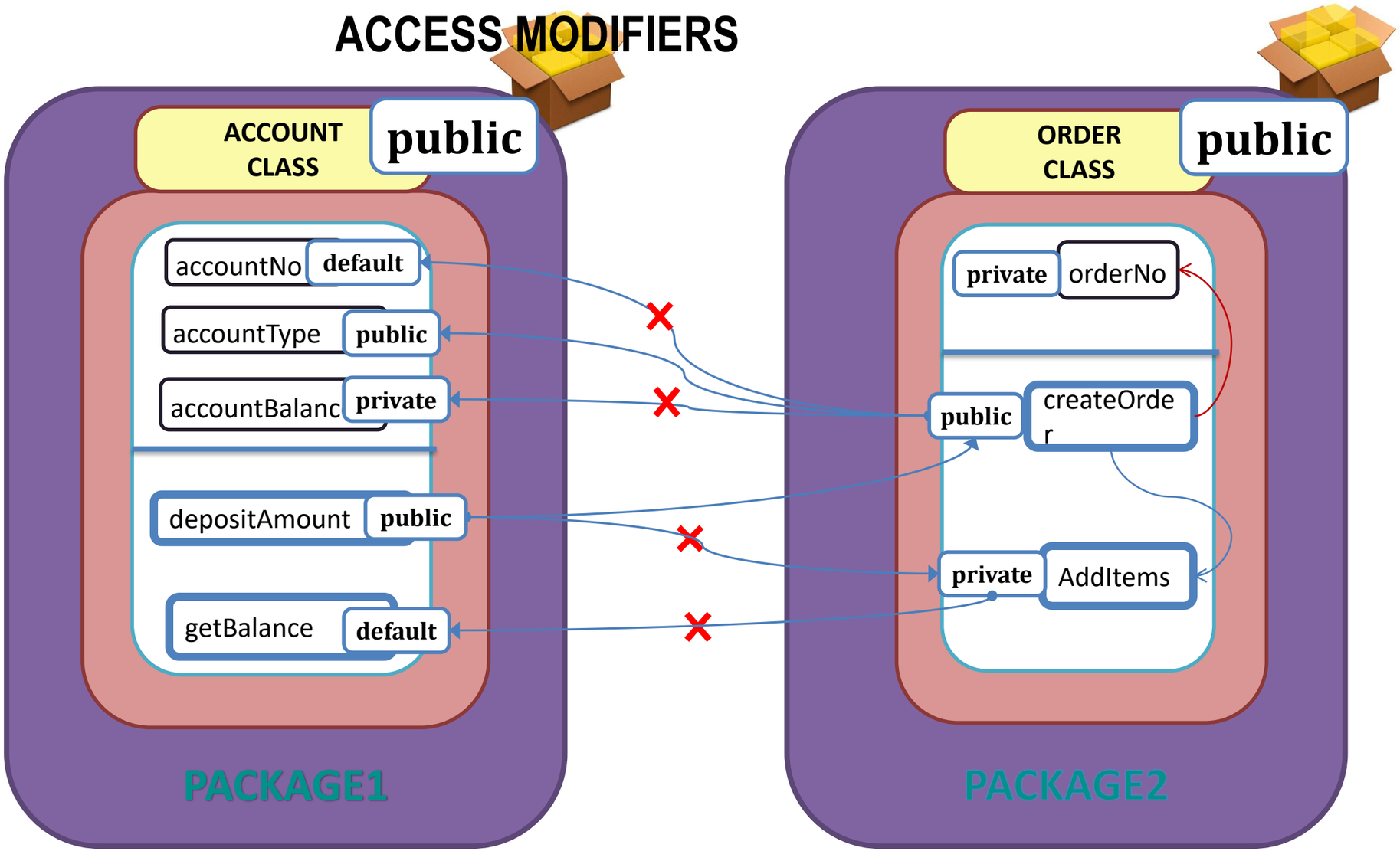
- private (-)
  - Declaration that is accessible only to the class/type in which it is declared

- default
  - Access to all classes/types within a package
  - Default declaration is done without specifying any modifier

ACCESS MODIFIERS



ACCESS MODIFIERS



ACCESS MODIFIER EXAMPLE

public class Employee
private instance variable empId
public method getEmpId

```
public class Employee {

    private int empId;

    public int getEmpId() {
        return empId;
    }}
```

```
class Employee {
   public static int
empcounter = 1000;
   private int empId;

   int getEmpId() {
        return empId;
```

default class Employee
public static variable empCounter
private instance variable empId