Class & Object

Tanjina Helaly

Content

- Initializing Fields/Instance variables.
- Scope & Lifetime of Variable.
- Arrays
- Parameter Passing
- Garbage Collection
- Package
- Access Modifier
- Recursion
- Variable Argument - vararg

Initializing Fields/Instance variables

Initializing Fields

- There are three ways in Java to give a field an initial value:
  - Direct Assignment
  - Instance Initialization Block
  - Constructors

1.Direct Assignment

```
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance = 100.0; // direct assignment

    // Methods
    public void deposit(double amount){
        balance = balance + amount;
    }

    public void withdraw(double amount){
        if (amount<balance)
            balance -= amount;
    }
}
```

2.Instance Initialization block

- Instance initialization blocks are indicated by blocks of code inside the class, but outside any method.
- *Whenever an object is created* from a class the code in *each instance initialization block is executed*.
- If there is more than one instance initialization block they are executed in order, from top to bottom of the class.
- Use initialization blocks *when the initialization cannot be done in a simple assignment and needs no extra input parameters*.
- Direct assignment and constructors are used far more often than initialization blocks.

2.Instance Initialization block

```
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance;

    // Methods
    public void deposit(double amount){
        balance = balance + amount;
    }
    public void withdraw(double amount){
        if (amount<balance)
            balance -= amount;
    }

    {   // Instance Initialization Block
        id = new Random().nextInt(99999) + "";
        balance = 100.0;
    }
}
```

3.Constructor

- A *constructor*
  - *Allocate space for instance variables.*
  - *initializes an object(its instance variables) immediately upon creation.*
- *Syntax:*
  - *It has the same name as* the class.
  - syntactically similar to a method.
    - Except has no return type. Not even **void**.
      - **This is because the implicit return type of a class' constructor is the class type itself.**

3.Constructor

- When called:
  - No explicit call
  - It is automatically called when the object is created, before the **new operator** completes.
- What should go inside Constructor
  - Normally the instance variables are initialized inside the constructor.

    Or
  - any set-up code

3.Constructor - Example

```
import java.util.Random;
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance;

    // Constructor without parameter
    public BankAccount(){
        id = new Random().nextInt(99999) + "";
        // name and balance will get default value
    }

    // Constructor with parameter
    public BankAccount(String _name, String _id, double _balance){
         name = _name;
         id = _id;
         balance = _balance;
    }

    public static void main(String[] args)
    {
        BankAccount ba = new BankAccount(“Rashid”, “1000500”, 1000.0);
    }
}
```

“THIS”KEYWORD

- refer to the *current object.*
- That is, **this is always a reference to the object on which the method was invoked.**
- **You can** use **this anywhere a reference to an object of the current class’ type is permitted.**

Constructor - Example

```
import java.util.Random;
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance;

    // Constructor without parameter
    public BankAccount(){
        id = new Random().nextInt(99999) + "";
        // name and balance will get default value
    }

    // Constructor with parameter
    public BankAccount(String _name, String _id, double _balance){
          name = _name;
          id = _id;
          balance = _balance;
    }

    public static void main(String[] args)
    {
        BankAccount ba = new BankAccount(“Rashid”, “1000500”, 1000.0);
    }
}
```

Constructor - Example

```
import java.util.Random;
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance;

    // Constructor without parameter
    public BankAccount(){
        id = new Random().nextInt(99999) + "";
        // name and balance will get default value
    }

    // Constructor with parameter
    public BankAccount(String name, String id, double balance){
         name = name;
         id = id;
         balance = balance;
    }

    public static void main(String[] args)
    {
        BankAccount ba = new BankAccount(“Rashid”, “1000500”, 1000.0);
    }
}
```

Constructor - Example

```
import java.util.Random;
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance;

    // Constructor without parameter
    public BankAccount(){
        id = new Random().nextInt(99999) + "";
        // name and balance will get default value
    }

    // Constructor with parameter
    public BankAccount(String name, String id, double balance){
        name = name;
        id = id;
        balance = balance;
    }

    public static void main(String[] args)
    {
        BankAccount ba = new BankAccount(“Rashid”, “1000500”, 1000.0);
    }
}
```

Variable on left is the same as right
What are we doing here
Which variables we are referring to

Constructor - Example

```
import java.util.Random;
public class BankAccount {
    // Instance variables
    public String name;
    public String id;
    public double balance;

    // Constructor without parameter
    public BankAccount(){
        id = new Random().nextInt(99999) + "";
        // name and balance will get default value
    }

    // Constructor with parameter
    public BankAccount(String name, String id, double balance){
        this.name = name;
        this.id = id;
        this.balance = balance;
    }

    public static void main(String[] args)
    {
        BankAccount ba = new BankAccount(“Rashid”, “1000500”, 1000.0);
    }
}
```

Scope & Lifetime of Variables

Scope of Variable

- What is scope?
  - A scope determines what variable are visible to other parts of your program. Or where the variable is accessible?
  - It also determines the lifetime of those variable.
- A block defines a *scope*.
  - the statements between opening and closing curly braces.
- **As a general rule**, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope.
- Within a block, variables can be declared at any point, but are valid only after they are declared.
  - Thus, if you define a variable at the start of a method, it is available to all of the code within that method.
  - Conversely, if you declare a variable at the end of a block, it is effectively useless, because no code will have access to it.

SCOPE OF VARIABLE - EXAMPLE

```
public void calculateInterest(double balance)
{
    if(balance > 10000)
    {
        float interest = 0.05f; // Scope of this variable is only inside the if
        block
    }
    else
    {
        interest = 0.02f; // compiler error. interest is declared inside the if
        block, hence can't access in else block
    }
}
```

- To make “interest” accessible to both if and else block it has to be declared outside of the block.

Scope of Variable - Example

```
public void calculateInterest(double balance)
{
    float interest; // accessible to anywhere inside the method.
    if(balance > 10000)
    {
        interest = 0.05f; // Ok
    }
    else
    {
        interest = 0.02f; // OK
    }
}
```

Scope of Variable

- Many other computer languages define two general categories of scopes:
  - global and local.
- However, these traditional scopes do not fit well with Java's strict, object-oriented model.
- In Java, the two major scopes are
  - those defined by a class and
  - those defined by a method.

When can 2 variables have same name

- Instance variable and Local variable.
- Local variable in 2 different methods.
- 2 Local variables in the same methods but only after the death of one Local variable.

OK	Wrong

public void calculateInterest(double balance) {
  if(balance > 10000){
    float interest = 0.05f; // OK
  }
  else {
    float interest = 0.02f; // OK
  }
}
	public void calculateInterest(double balance) {
	  float interest;
	  if(balance > 10000){
	    float interest = 0.05f; // Compiler error
	  }
	  else {
	    interest = 0.02f; // OK
	  }
	}

Arrays

Array Again

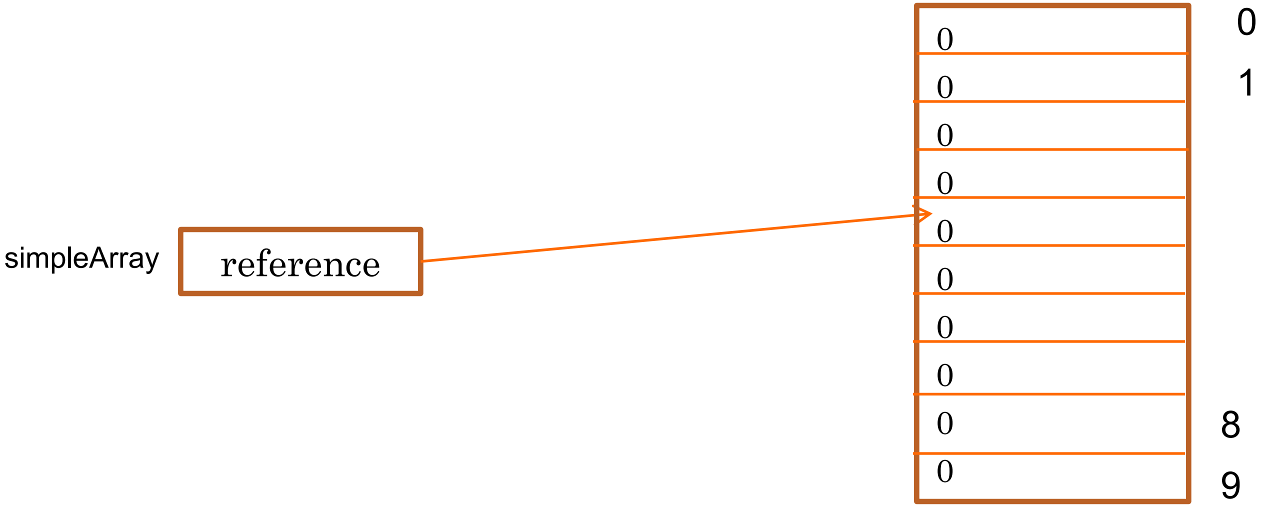
- What is the “new” keyword during array creation.
- Is Array variable reference type?

```
int[] sampleArray = new int[10];
```

Array Again

- What is the “new” keyword during array creation.
- Is Array variable reference type?

```
int[] sampleArray = new int[10];
```

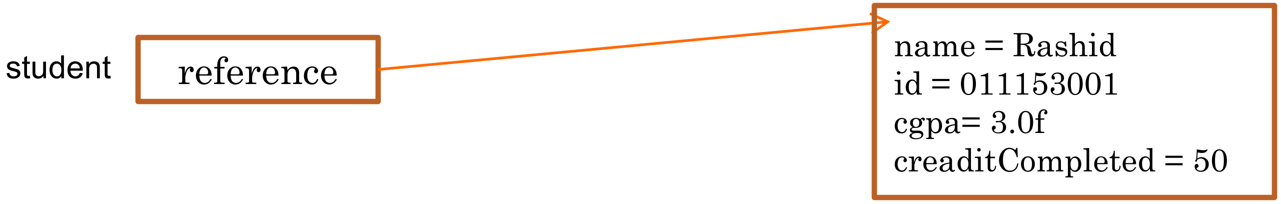


Array Again

- When an array is created, each element will be initialized to its default value.
- What is the initial value for each of the element of the arrays below.
  - int[] sampleArray = new int[10];
  - Student[] students = new Student[10];

Reference Type with Not Null

Student student = new Student(“Rashid”, “011153001”, 3.0f, 50);

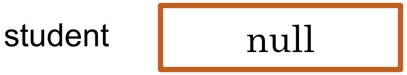


name = Rashid
id = 011153001
cgpa= 3.0f
creaditCompleted = 50

- What value will you get when you access the following attributes of student reference variable/object.
  - student.name
  - student.id
  - student.cgpa
  - student.creditCompleted

Reference Type with null

```
Student student = null;
```



student

null

- What value will you get when you access the following attributes of student reference variable/object.
  - student.name
  - student.id
  - student.cgpa
  - student.creditCompleted

Reference Type with null

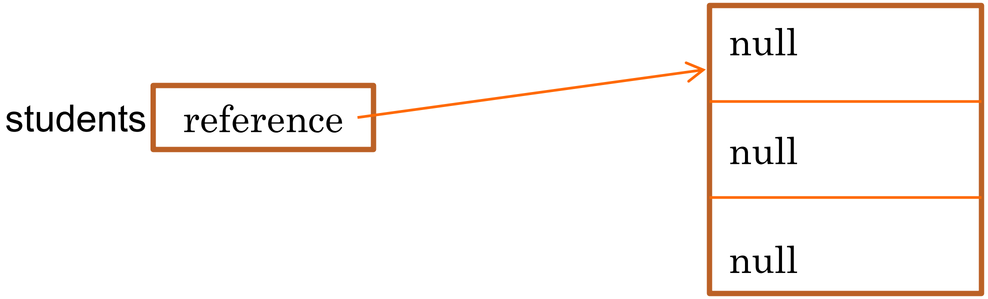
- We cant access any member via the reference variable when no object is created.
- Accessing the member will throw NullPointerException.

Reference Type Array

- Example

  Student[] students = new Student[3];

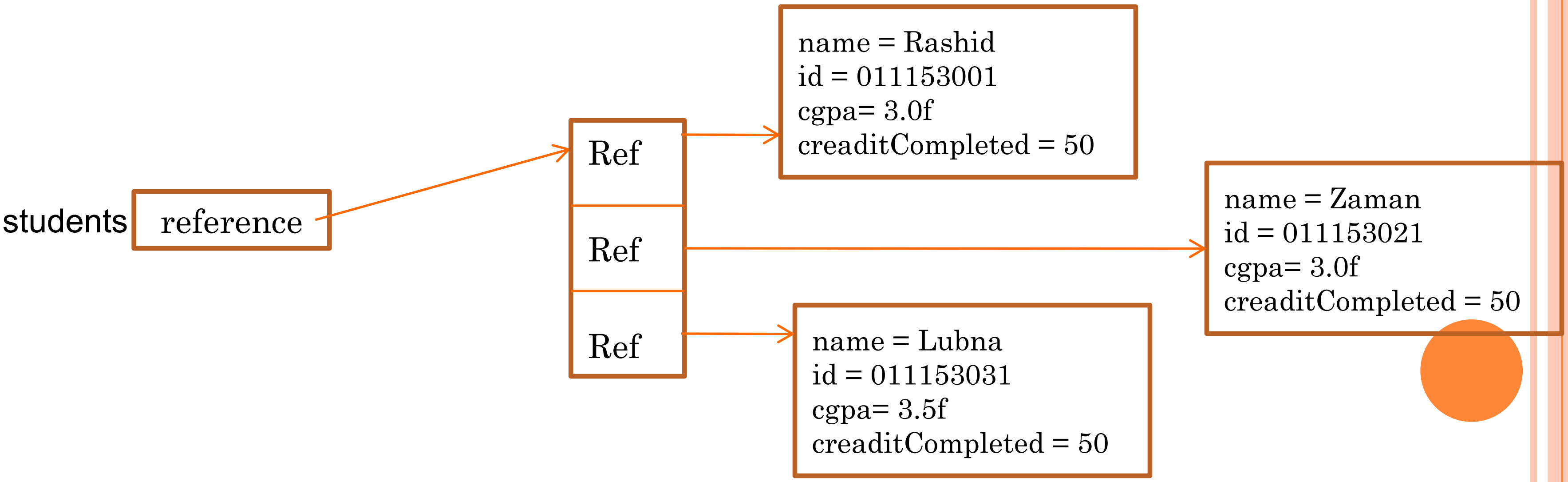
  System.out.println(students[0].cgpa); // What would be the output of this line.



Reference Type Array

- Need to initialize the element before accessing.
- Example

```
 Student[] students = new Student[3];
students[0] = new Student(“Rashid”, “011153001”, 3.0f, 50);
students[1] = new Student(“Zaman”, “011153021”, 3.0f, 50);
students[2] = new Student(“Lubna”, “011153031”, 3.5f, 50);
System.out.println(students[0].cgpa); // What would be the output of this line.
```



Parameter Passing

Parameter Passing

- 2 different ways
  - Pass By Value
  - Pass By Reference

- In **Java** all parameters are **passed by value**

Pass by Reference

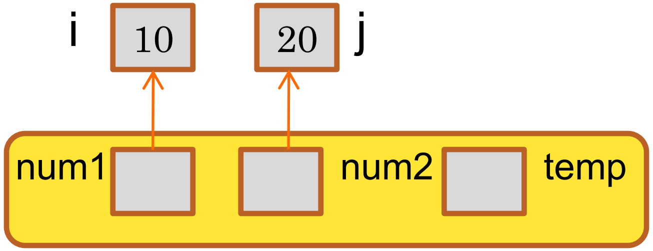
- Send the location of the parameter.
- Original value chage
- C, C++, php

```
main() {
    int i = 10, j = 20;
    cout << i << " " << j << endl;
    swapThemByRef(i, j);
    cout << i << " " << j << endl; // displays 20 10 ...
}
void swapThemByRef(int& num1, int& num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
 }
```

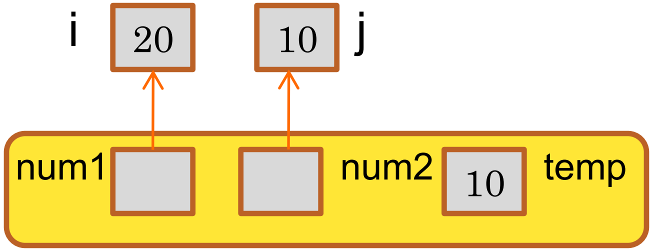
```
Output:
      10 20
      20 10
```

Pass by Reference

- At the beginning of function call



- After the function execution.



PASS BY VALUE

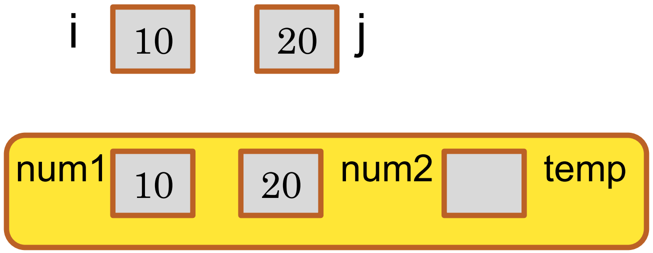
- Send a copy of the original parameter.
- Original value does not change
- Example:

```
main() {
    int i = 10, j = 20;
    cout << i << " " << j << endl;
    swapThemByVal(i, j);
    cout << i << " " << j << endl; // displays 20 10 ...
}
void swapThemByVal(int num1, int num2) {
    int temp = num1;
    num1 = num2;
    num2 = temp;
 }
```

```
Output:
      10 20
      10 20
```

Pass by Value

- At the beginning of method call



i 10

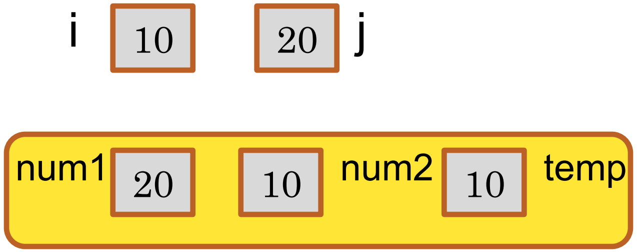
20 j

num1 10

20 num2

temp

- After the method execution.



i 10

20 j

num1 20

10 num2

10 temp

Pass by Value

- Java Example:

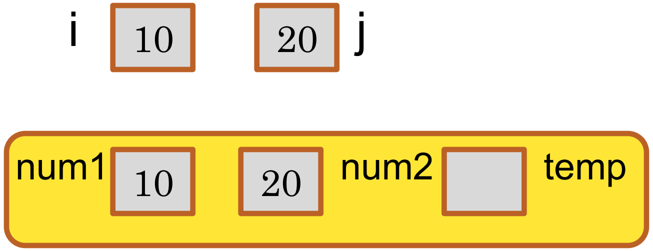
```
public class PassByValue {
     public static void main(String[] args) {
         int a=10, b=20;
         System.out.printf("a-%d:b-%d\n", a, b);
         swapThemByVal(a, b);
         System.out.printf("a-%d:b-%d\n", a, b);
     }

     static void swapThemByVal(int num1, int num2) {
          int temp = num1;
          num1 = num2;
          num2 = temp;
     }
}
```

Output:

```
     a-10:b-20
     a-10:b-20
```

Pass by Value

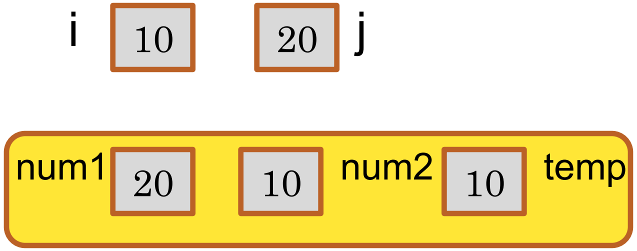
- At the beginning of method call



i 10 20 j

num1 10 20 num2 temp

- After the method execution.



i 10 20 j

num1 20 10 num2 10 temp

Pass by Value – with Object

- When an object is passed to a method, the situation changes dramatically,
- Java will pass by value but the effect will be like pass-by-reference.
- Why?
  - When we create a variable of a class type, it will store the reference to an object.
  - Thus, when the value of the variable will be passed to a method, it will pass the reference of the same object.
  - This effectively means that objects act as if they are passed to methods by use of pass-by-reference.
  - Changes to the object inside the method *do affect the object used as an argument*.

Pass by Value – with Object

```
public class Test{
    String testName;
    float score;

    Test(String n, float s){
        testName = n;
        score = s;
    }

    void display(){
       System.out.printf("TestName: %s ; Score: %.2f\n", testName,
       score);
    }
}
```

Pass by Value – with Object

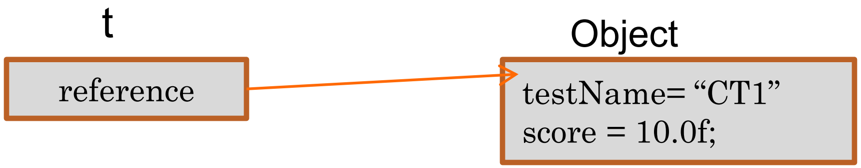
```
public class PassByValue {
    public static void main(String[] args) {
        Test t = new Test("CT1", 10);
        t.display();
        updateScore(t, 15.0f);
        System.out.println("After Update:");
        t.display();
    }

    static void updateScore(Test test, float newScore) {
        test.score = newScore;
    }
}
```

Output:

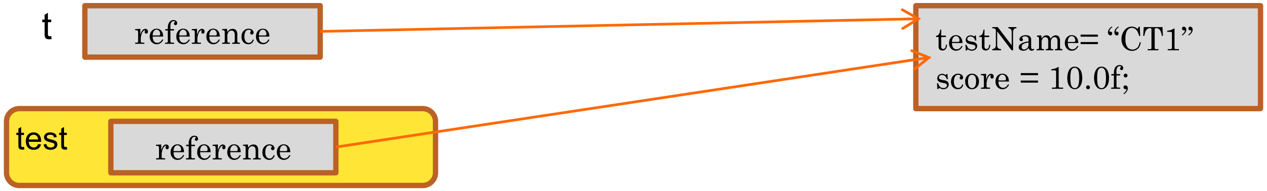
```
TestName: CT1 ; Score: 10.00
After Update:
TestName: CT1 ; Score: 15.00
```

Pass by Value – with Object

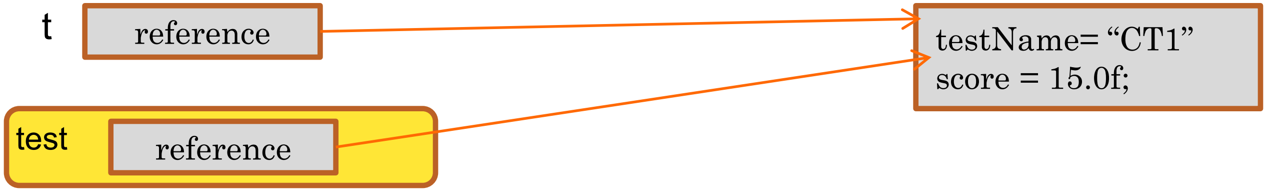
- Before the method call



- At the beginning of method call
  - Both “t” and “test” are referring to the same object.
  - Updating the object using any variable will be reflected in the other one.



- Just at the end of method execution – before exiting the method



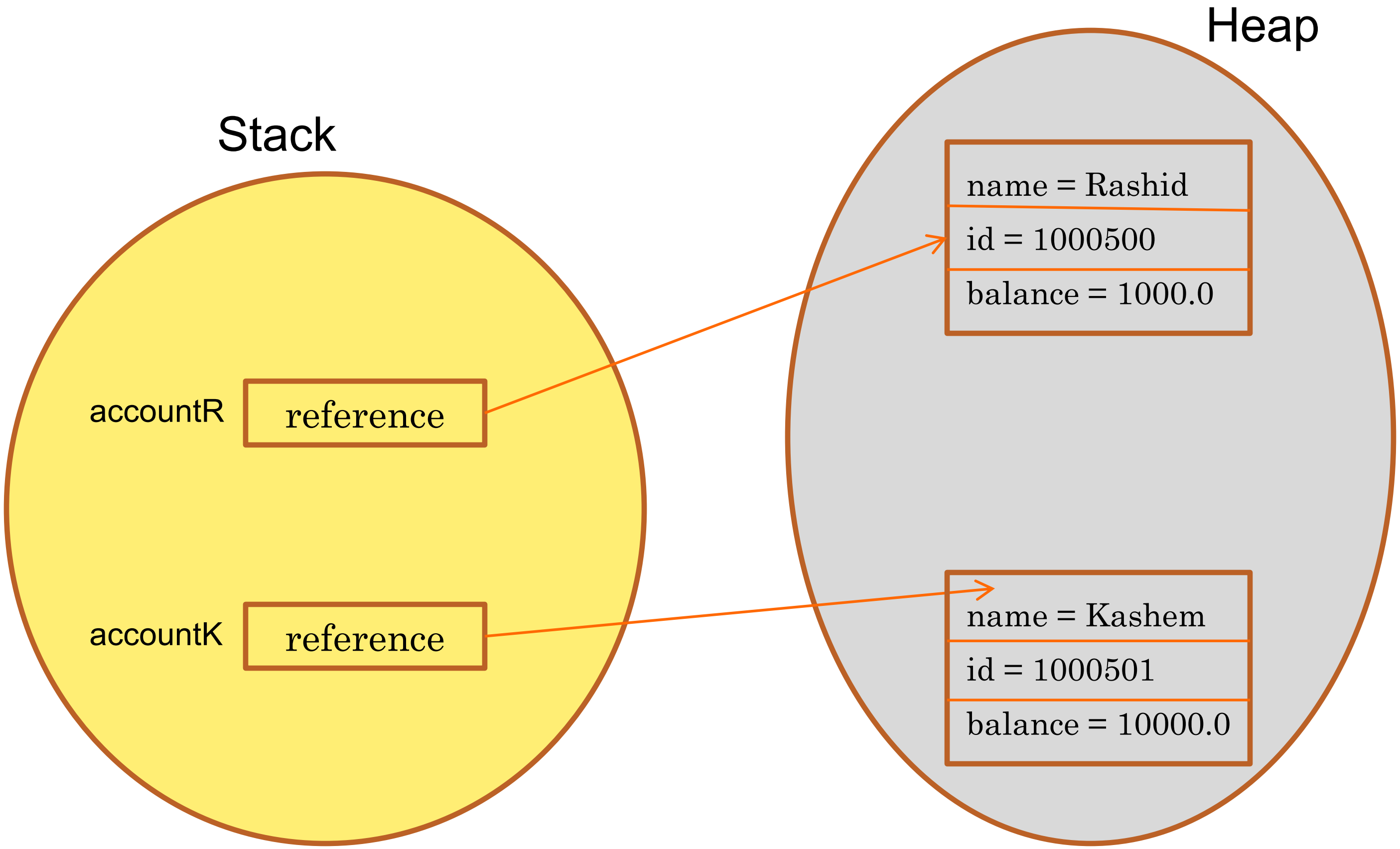
Garbage Collection

Garbage Collection

- when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- Garbage collection only occurs sporadically (if at all) during the execution of your program.
- Different Java run-time implementations will take varying approaches to garbage collection,
- but for the most part, you should not have to think about it while writing your programs.

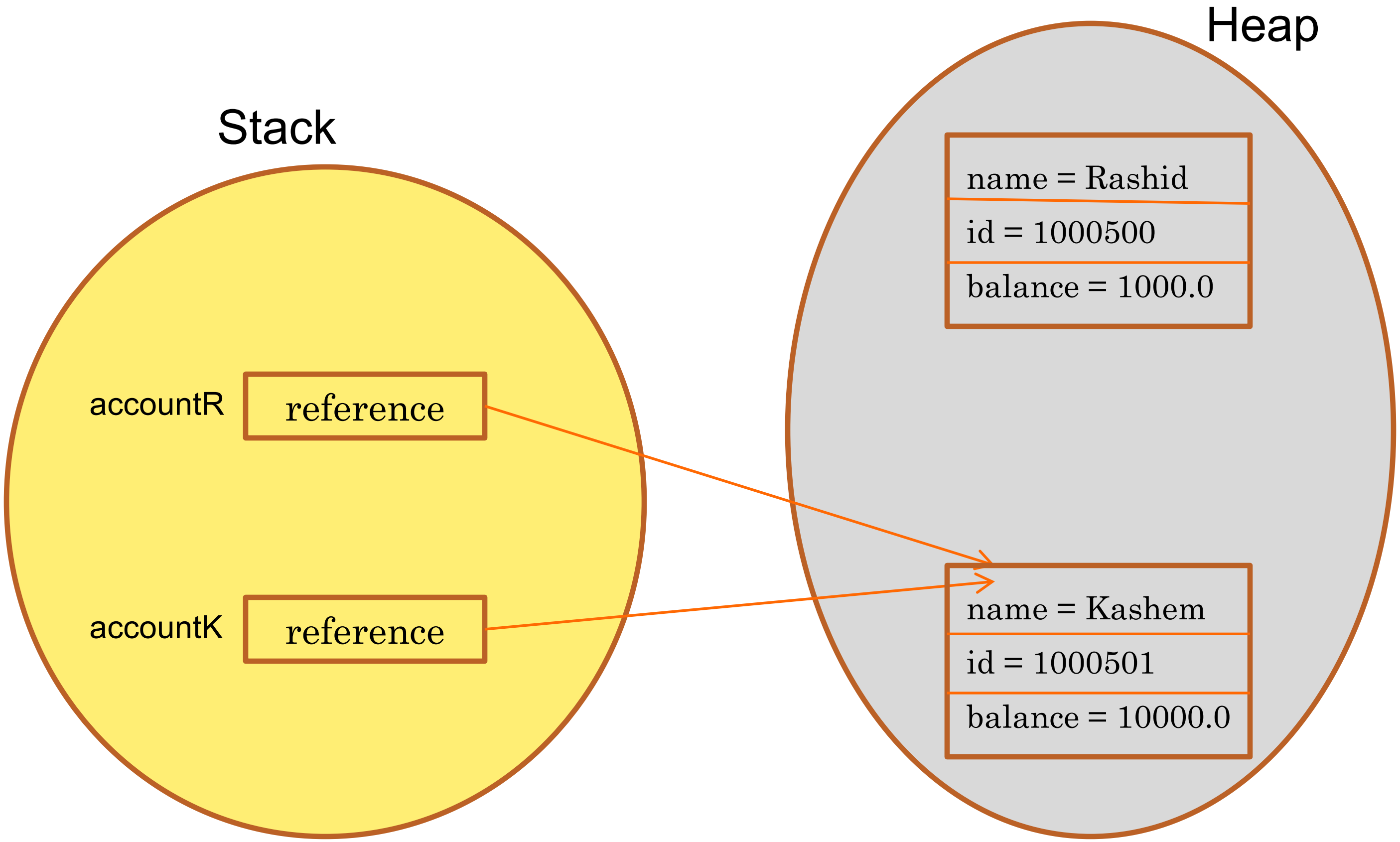
Garbage Collection – scenario#1

BankAccount accountR = **new BankAccount(“Rashid”, “1000500” , 1000.0);**
BankAccount accountK = **new BankAccount((“Kashem”, “1000501” , 10000.0);**



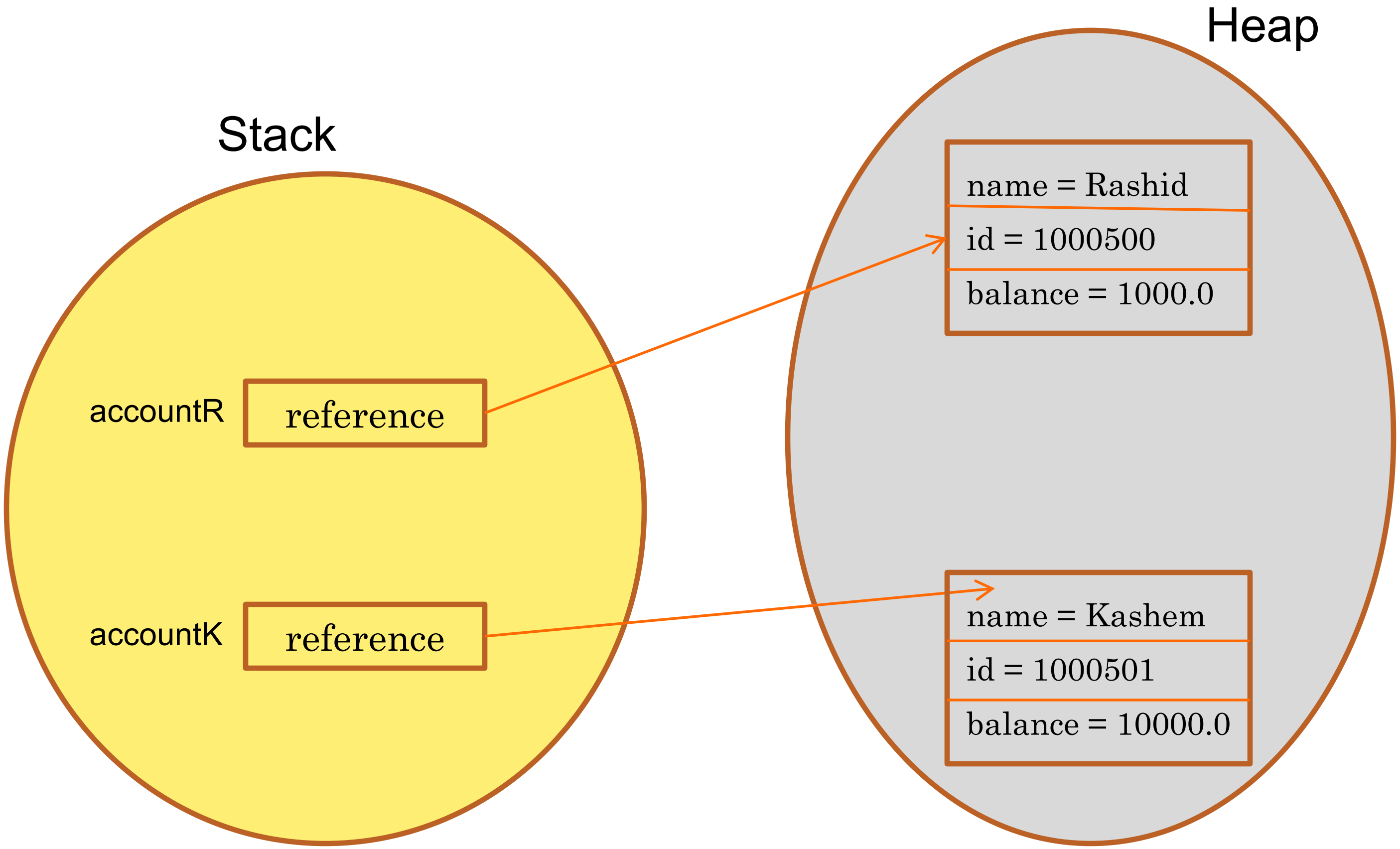
GARBAGE COLLECTION – SCENARIO#1

BankAccount accountR = **new BankAccount(“Rashid”, “1000500” , 1000.0);**
BankAccount accountK = **new BankAccount((“Kashem”, “1000501” , 10000.0);**
accountR = accountK;
// Rashid’s object can no longer be accessed and is eligible garbage collection



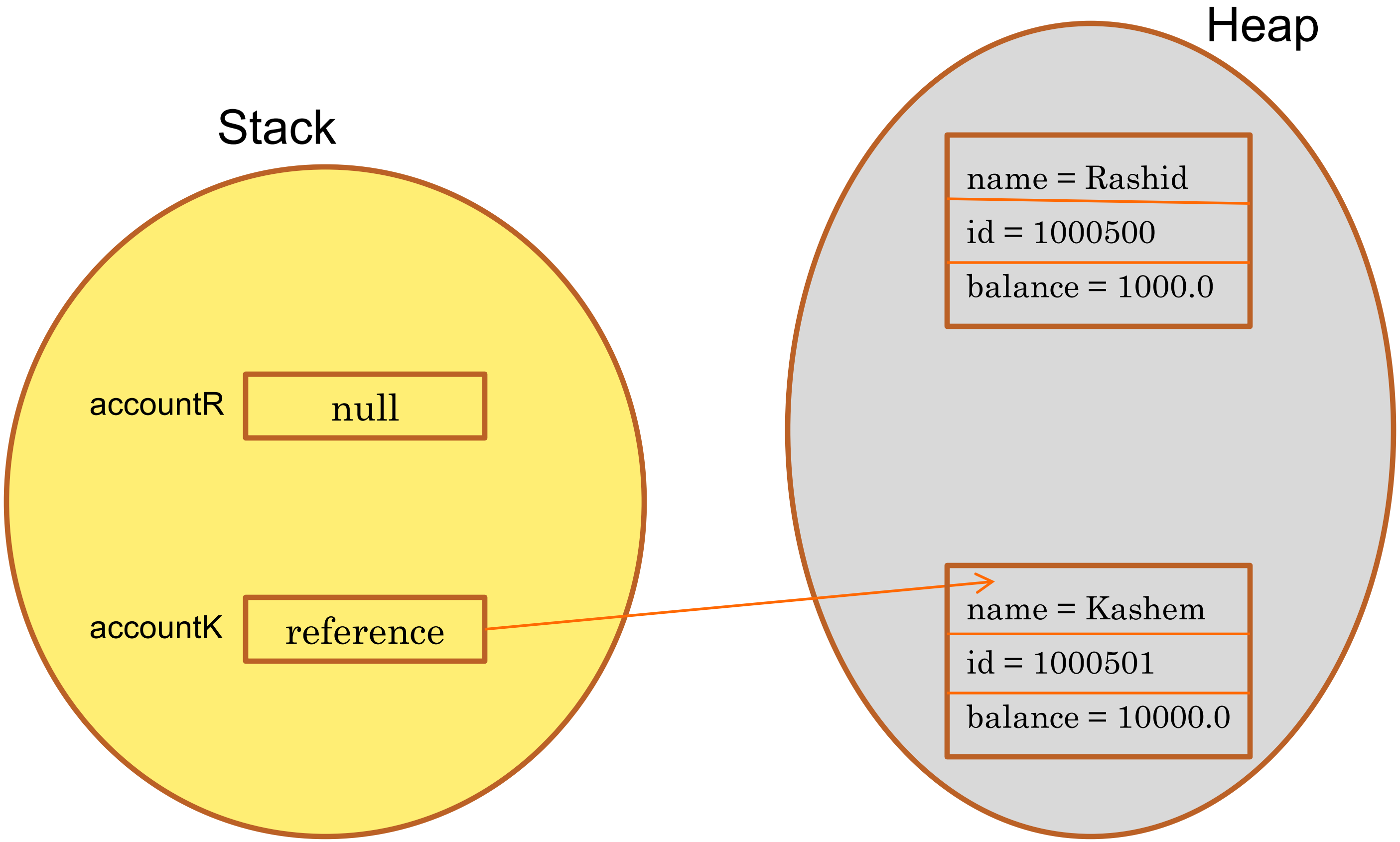
Garbage Collection – scenario#2

BankAccount accountR = **new BankAccount(“Rashid”, “1000500” , 1000.0);**
BankAccount accountK = **new BankAccount((“Kashem”, “1000501” , 10000.0);**



GARBAGE COLLECTION – SCENARIO#2

BankAccount accountR = **new BankAccount(“Rashid”, “1000500” , 1000.0);**
BankAccount accountK = **new BankAccount((“Kashem”, “1000501” , 10000.0);**
accountR = null;
// Rashid’s object can no longer be accessed and is eligible garbage collection

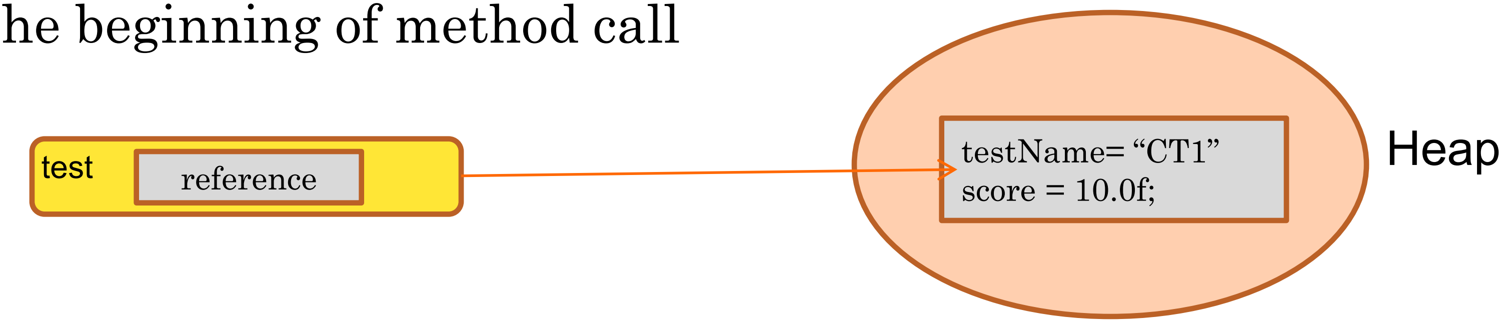


Garbage Collection – scenario#3

```
public class TestMain{
    public static void main(String[] args) {
        updateScore(new Test("CT1", 10), 15.0f);
    }

    static void updateScore(Test test, float newScore) {
        test.score = newScore;
    }
}

public class Test{
    String testName;
    float score;

    Test(String n, float s){
        testName = n;
        score = s;
    }
}
```

GARBAGE COLLECTION – SCENARIO#3

- At the beginning of method call



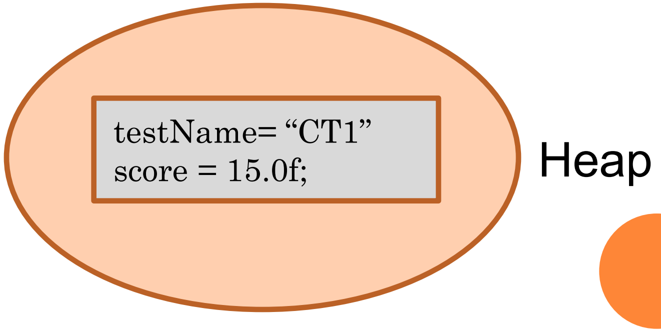
test

reference

testName= “CT1”
score = 10.0f;

Heap

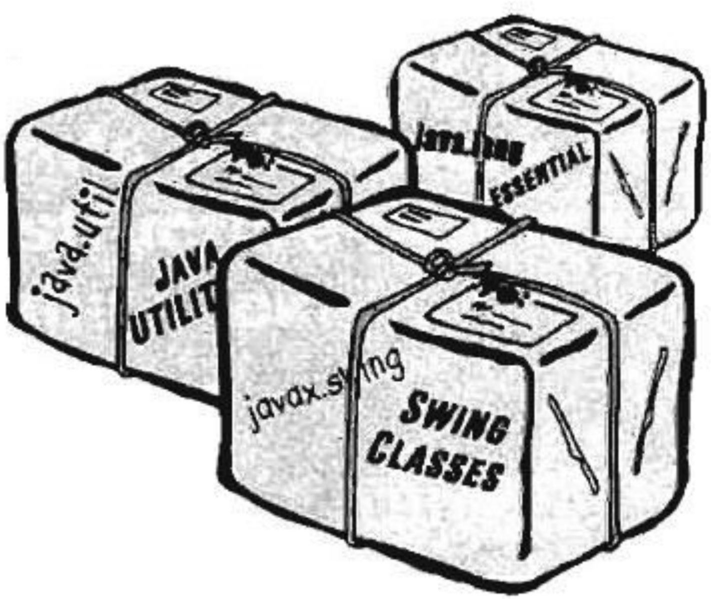
- After exiting the method.
  - “test” variable will no longer be available.
  - The Test object can no longer be accessed and is eligible garbage collection



testName= “CT1”
score = 15.0f;

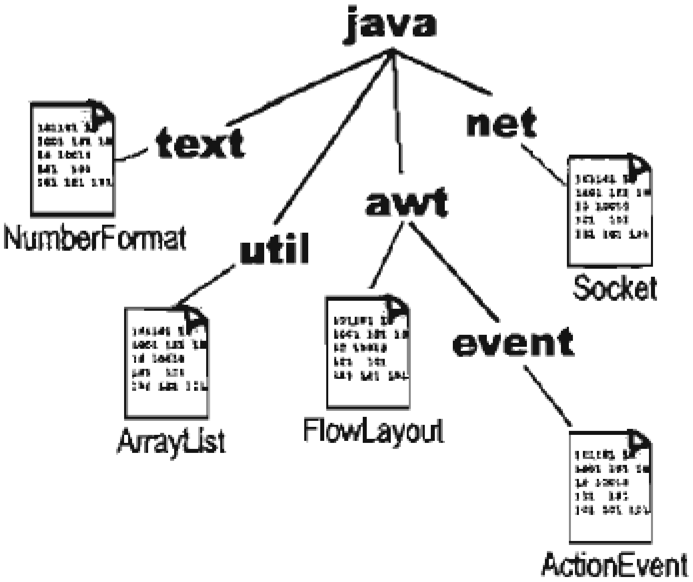
Heap

PACKAGE


java.util
JAVA UTILITY
java.lang
ESSENTIAL
javax.swing
SWING CLASSES

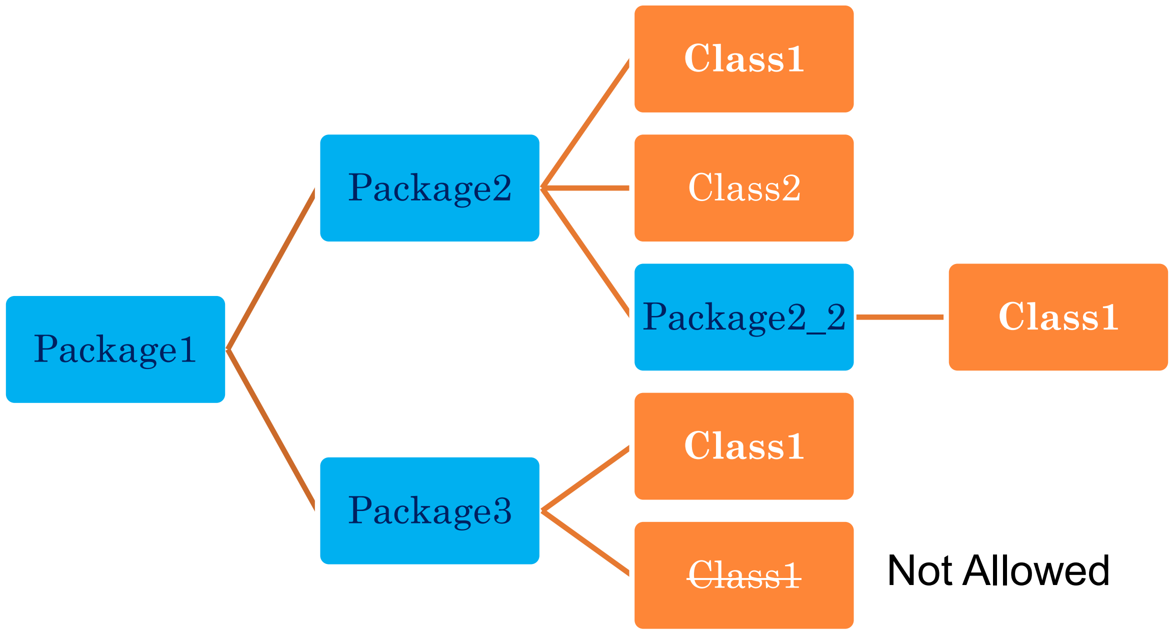
WHAT IS PACKAGE?

- Packages are used to group related classes.
- A package is a namespace that organizes a set of related classes and interfaces.
- Conceptually you can think of packages as being similar to different folders on your computer.



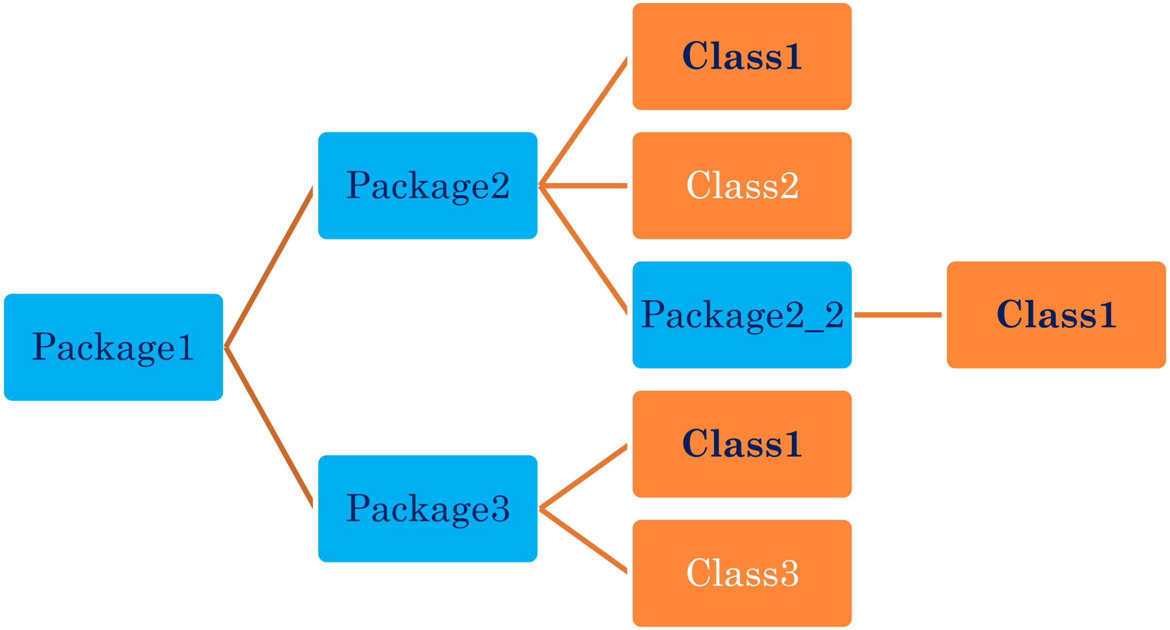
What is Package?

- Classes in same package can not have duplicate name.
- Classes in different packages can have the same name.



WHAT IS PACKAGE?

- Classes in same package can not have duplicate name.
- Classes in different packages can have the same name.



HOW TO CREATE PACKAGE?

- To create a package is quite easy:
  - simply include a **package command as the first statement** in a Java source file.

    package *pkg;*
  - Any classes declared within that file will belong to the specified package.
  - Java uses **file system directories to store packages**.
- You can create a hierarchy of packages.
  - Use period/dot to separate each package name from the one above it.

    package *pkg1.pkg2.pkg3;*

Package -Example

- Example:

```
package uiu.cse;
public class Test{
   public void display() {
      System.out.println( "Hello for Test class." );
  }
}
```

1. The class must be in a file named “Test.java"
2. Place the file “Test.java" in a directory called “cse"
3. Place directory “cse" in a directory called “uiu”.
4. The directory “cse" can be placed anywhere, but you need to set the classpath.

PACKAGE -EXAMPLE

- If you use IDE,
  - 1-3 will be done automatically and “cse” will be placed under the “src” folder.
  - If no package is specified the file will be placed in a default package which maps to “src” folder

Benefits of using Package

- The package is both a naming and a visibility control mechanism.
- Packages are important for three main reasons.
  - **First**, they help the overall organization of a project or library.
    - Can organize code in a logical manner
    - makes large software projects easier to manage

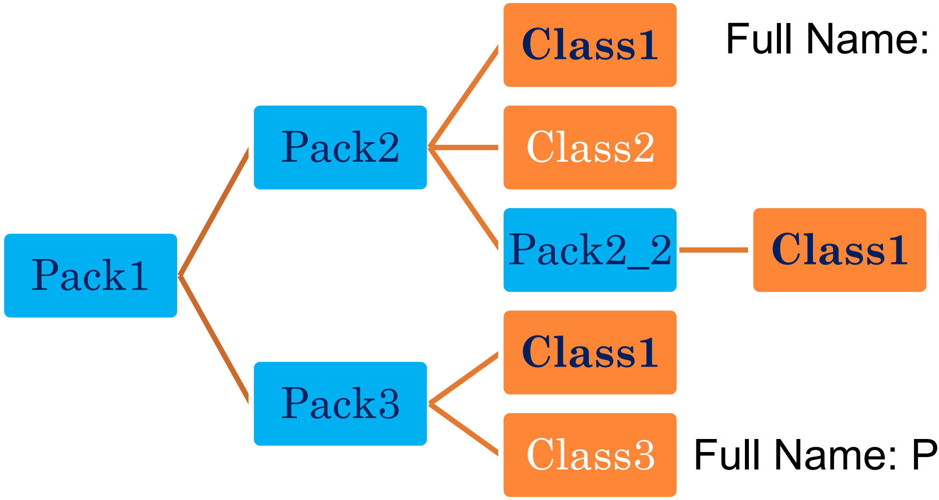
Benefits of using Package

- **Second**, packages give you a name scoping, to help prevent collisions.
  - What will happen if you and 12 other programmers in your company all decide to make a class with the same name.

- **Third**, packages provide a level of security,
  - Can define **classes** inside a package that are not accessible by code outside that package.
  - can also define class **members** that are exposed only to other members of the same package.

Class's Full Name

- A class has a full name, which is a combination of the package name and the class name.



Pack1

Pack2

Class1

Class2

Pack2_2

Class1

Pack3

Class1

Class3

Full Name: Pack1.Pack2.Class1

Full Name: Pack1.Pack2.Pack2_2.Class1

Full Name: Pack1.Pack3.Class3

How to access Class

- To use a class in same package, you can use the class name (short name).
- To use a class in a different package, you must tell Java the full name of the class.
  - You use either an Import statement at the top of your source code, and use short name or
  - you can type the full name every place you use the class in your code.

Class in Same Package

- Test Class

```
package uiu.cse;
public class Test{
   public void display() {
      System.out.println( "Hello for Test class." );
   }
}
```

- Main Class

```
package uiu.cse;
public class TestMain {
   public static void main(String[] args) {
      Test test = new Test();
      test.display();
   }
}
```

Class in Different Package – Full Name

- Test Class

```
package uiu.cse;
public class Test{
    public void display() {
        System.out.println( "Hello for Test class." );
    }
}
```

- Main Class

```
package uiu.cse.test; // any package other than uiu.cse

public class TestMain {
    public static void main(String[] args) {
        uiu.cse.Test test = new uiu.cse.Test(); // use Full name
        test.display();
    }
}
```

CLASS IN DIFFERENT PACKAGE - IMPORT

- Test Class – see previous slide
- Main Class

```
import uiu.cse.Test; // Need to import the class.
public class TestMain {
    public static void main(String[] args) {
       Test test = new Test(); // use Short name
       test.display();
   }
}
```

  - To import all classes under a package we need to use * after the package name. Example:

```
import uiu.cse.*;
```

- **java.lang.*** package is always implicitly get imported by Java. [explicit import is not needed for this package]

ACCESS CONTROL/ACCESS MODIFIER

- How a member can be accessed is determined by the *access modifier attached to its* declaration.
- 4 types
  - public
  - Protected
  - Default/Package Access – No modifier
  - Private
- Outer class can only be declared as public or default.

Private

- Members declared private are accessible only in the class itself
- Example:

```
class Private {
    private String name = "Private";
    public void print() {
      System.out.println( name );
    }
}
class PrivateExample {
    public static void main( String[] args ) {
        Private pr = new Private();
        pr.print(); // OK
        System.out.println( pr.name ); // Compile error
    }
}
```

Default/Package Access

- When no access modifier is specified
- Accessible in the package that contains the class
- Not accessible outside the package that contains the class.
    - Not even child class.

Default/Package Access - Example

Class that will be accessed from other classes

```
package test;
class Default{
      String name = “Default“;
}
```

Class under different package

```
package test1;
class DefaultExample {
      public static void main( String[] args ) {
            Default dfl= new Default();
            System.out.println( dfl.name ); // Compile error
      }
}
```

Class under same package

```
package test;
class DefaultExample1 {
      public static void main( String[] args ) {
            Default dfl = new Default();
            System.out.println( dfl.name ); // OK
      }
}
```

Default/Package Access - Example

Child Class under different package

```
package test1;
class DefaultChild extends Default {
      public DefaultChild() {
           name = “Child”; // Compile error
      }
}
```

Child Class under same package

```
package test;
class DefaultChild1 extends Default {
      public DefaultChild1() {
           name = “Child”; // OK
      }
}
```

Protected

- Members declared protected are directly accessible to any subclasses,
  - Even if the child is in different package
- directly accessible by code in the same package.

Protected - Example

Class that will be accessed from other classes

```
package test;
class Protected {
    protected String name = "Protected“;
}
```

Class Under Different Package

```
package test1;
class ProtectedExample {
    public static void main( String[] args ) {
        Protected pr = new Protected();
        System.out.println( pr.name ); // Compile error
        }}
```

Class Under Same Package

```
package test;
class ProtectedExample1 {
    public static void main( String[] args ) {
        Protected pr = new Protected();
        System.out.println( pr.name ); // OK
    }}
```

PROTECTED - EXAMPLE

Child Class Under Different Package

```
package test1;
class ProtectedChild extends Protected {
    public ProtectedChild() {
        pr.name = “Child”; // OK
    }
}
```

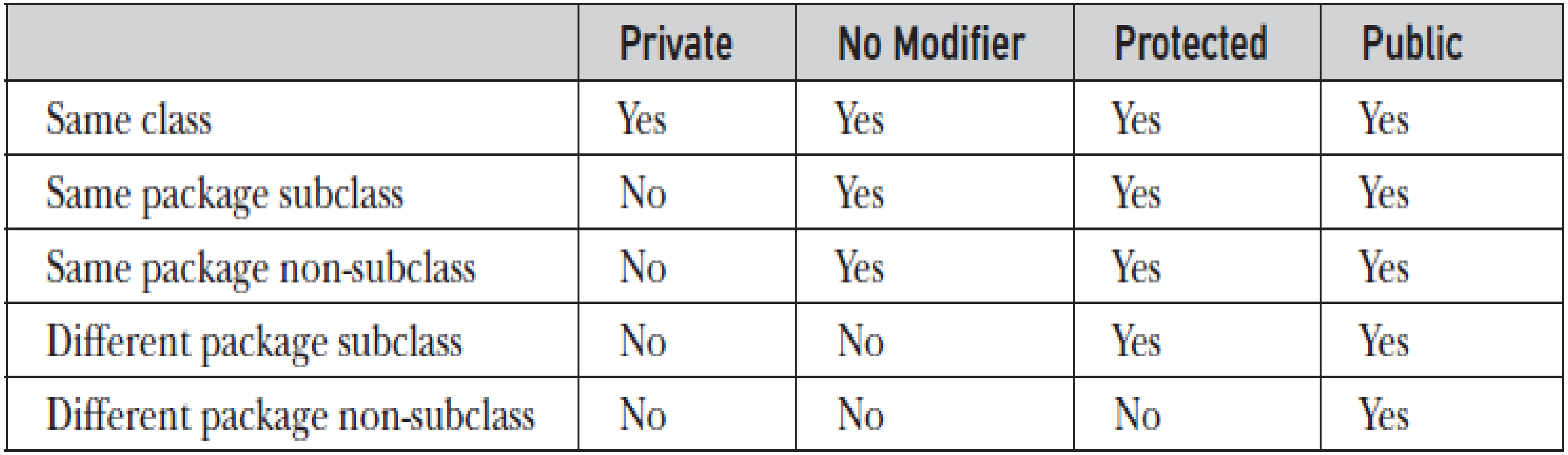
Child Class Under Same Package

```
package test;
class ProtectedChild1 extends Protected {
    public ProtectedChild1() {
        pr.name = “Child”; // OK
    }
}
```

Public

- Members declared public are accessible anywhere the class is accessible
- All the scenario described in other 3 types of access modifier won’t give any compile error

Access Modifier Chart



	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Recursion

- It is the process of a method calling itself. A recursion has two parts
  - stop condition
  - call itself
- Recursion is an expensive procedure and it is better to avoid recursion if you do not need.
- Example: Factorial, Fibonacci

```
public class MyMathLibrary {
    public static int factorial(int n){
        if (n <= 0)
            return 1; // stop condition
        return n * factorial(n-1); // call itself
    }
    public static void main( String[] args ){
      System.out.print("Factorial of 5 is: " + factorial(5));
      System.out.print("Factorial of 6 is: " + factorial(6));
    }
}
```

Variable Argument

VARIABLE ARGUMENT – OLD APPROACH

```
// Use an array to pass a variable number of arguments to a method.
// This is the old-style approach to variable-length arguments.
class PassArray {
    static void vaTest(int v[]) {
        System.out.print("Number of args: " + v.length + " Contents: ");
        for(int x : v)
          System.out.print(x + " ");
        System.out.println();
    }
    public static void main(String args[]) {
        // Notice how an array must be created to hold the arguments.
        int n1[] = { 10 };
        int n2[] = { 1, 2, 3 };
        int n3[] = { };
        vaTest(n1); // 1 arg
        vaTest(n2); // 3 args
        vaTest(n3); // no args
    }
}
```

Output:

```
Number of args: 1 Contents: 10
Number of args: 3 Contents: 1 2 3
Number of args: 0 Contents:
```

Variable Argument – New Approach

- Introduced in Java 5.
- Can pass variable number of argument.
- This feature is called ***varargs*** - short for *variable-length arguments*.
- *A method that takes a variable number of arguments is* called a *variable-arity method, or simply a varargs method.*
- A variable-length argument is specified by three periods (**...).**
- Example : static void vaTest(int ... v) {
- The argument v act as an array

Vararg - example

```
// Demonstrate variable-length arguments.
class VarArgs {
 // vaTest() now uses a vararg.
 static void vaTest(int ... v) {
       System.out.print("Number of args: " + v.length + " Contents: ");
       for(int x : v)
          System.out.print(x + " ");
       System.out.println();
 }
 public static void main(String args[]) {
       // Notice how vaTest() can be called with a
       // variable number of arguments.
       vaTest(10); // 1 arg
       vaTest(1, 2, 3); // 3 args
       vaTest(); // no args
 }
}
```

Output: Same as before

Variable Argument

- A method can have “normal” parameters along with a variable-length parameter.
- However, the variable-length parameter must be the last parameter declared by the method.
- For example,
  - Following method declaration is perfectly acceptable:

    int doIt(int a, int b, double c, int ... vals) {
  - Inacceptable method

    int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!
  - Can’t have more than one vararg parameter

    int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

Reference

- Java:Complete Reference Chapter 6,7,9