Classes and Methods

CS 180

Sunil Prabhakar

Department of Computer Science

Purdue University

Objectives

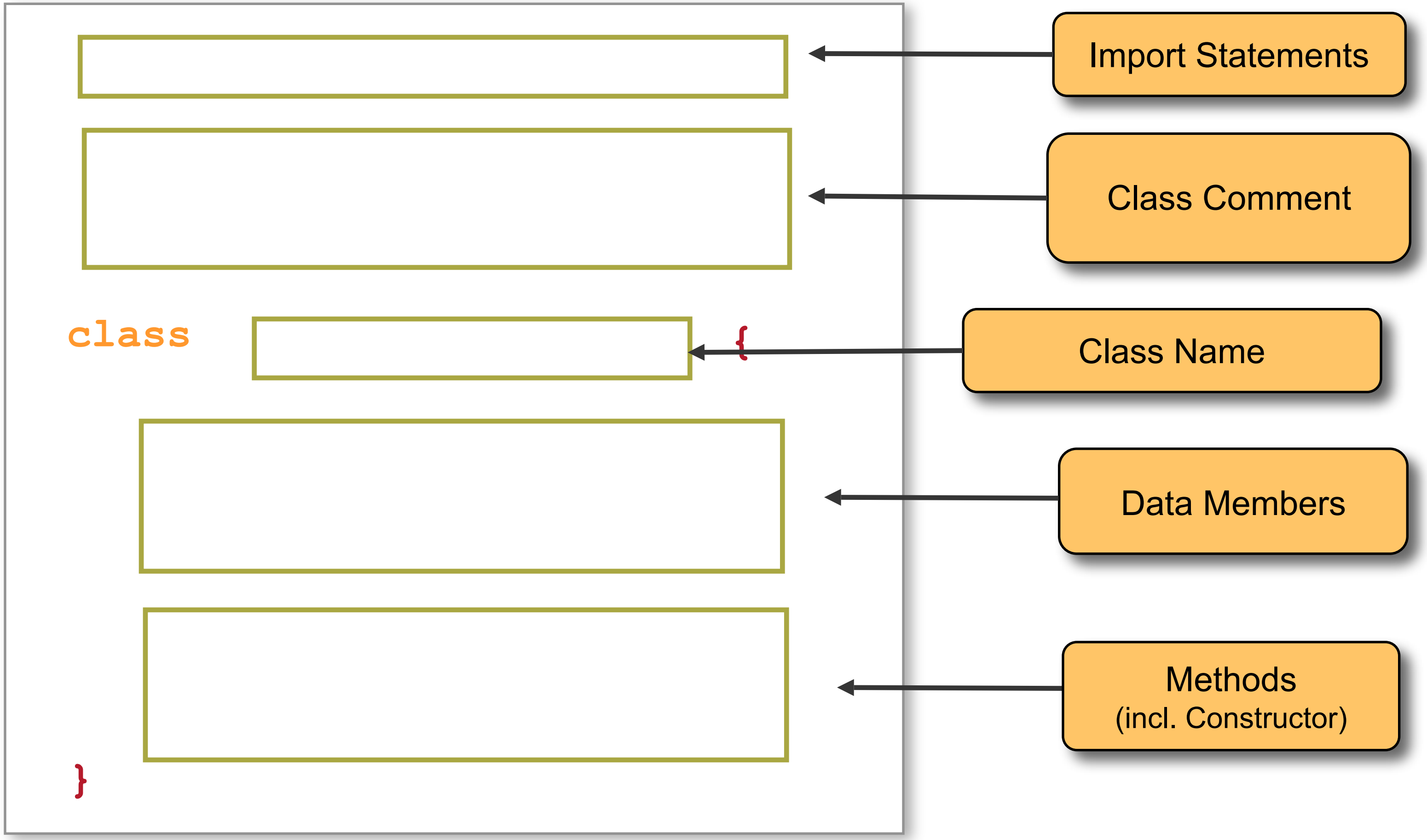
Review

- Methods
  - Constructors
  - Call-by-value
  - Overloading
- Private and public modifiers
- Scope and extent
- **this** keyword
- Static methods and data

User-defined Classes

- Create a class whenever no existing class fits our needs.
- Data members
  - each objects gets its own copy
- Methods
  - only methods defined for a class can be called on an object of that class (encapsulation).

Template for Class Definition

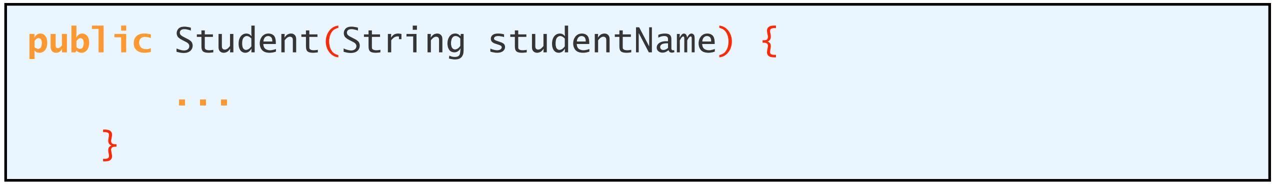


Student

```java
import javax.swing.*;
/************************************/
/* Java class for a single student  */
/* Author: Sunil Prabhakar          */
/* Date: September 8, 2009          */
/************************************/
class Student {
   private String name;
   private String id;

   public Student(String studentName){
      name = studentName;
      id = “”;
   }
   public void setName(String studentName){
      name = studentName;
   }
   public String getName(){
      return name;
   }
   public String getId(){
      return id;
   }
   public void setId(String studentId){
      id = studentId;
   }
}
```

Constructors

- Special type of method.
- Called whenever a new object is created.
- Special syntax:
  - name is same as class name;
  - called using **new** ClassName(…);
  - no return type (or return statement);



```
public Student(String studentName) {
      ...
   }
```

- If none defined, compiler adds a default one (with no parameters)

Call-by-Value

- When a method is called:
  - temporary memory space is created for the method
    - parameters
    - local data
  - Passed arguments are copied to corresponding parameters
    - left-to-right association
    - must be assignment-compatible
    - pass-by-value; call-by-value
  - method execution begins

Call-by-Value Example

```
class MyClass {
   public double myMethod(int one, double y ) {
      int i=5;
      one += 6;
      i *= y;
      return i;
    }
}
```

```
MyClass myObj;
int x, y;
myObj = new MyClass();
x = 10;
y = 20;
y = (int)myObj.myMethod(x, y);
System.out.println(x + " " + y);
```

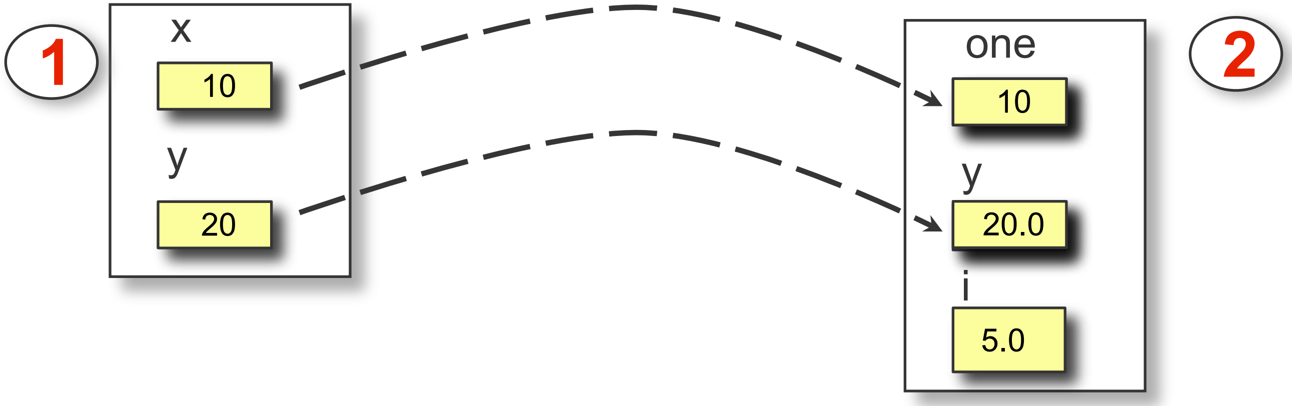
Memory Allocation for Parameters

```
MyClass myObj;
int x, y;
myObj = new MyClass();
x = 10;
y = 20;
y = (int) myObj.myMethod(x, y);
System.out.println(x + " " + y);
```

1

2

```
class MyClass {
   public double myMethod(int one, double y ){
      int i=5;
     one += 6;
      i *= y;
      return i;
    }
}
```



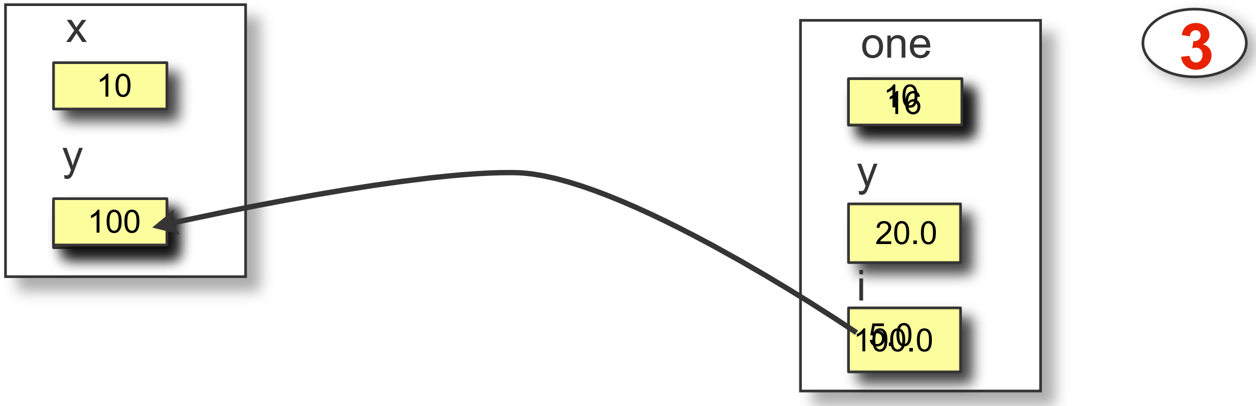
Memory Allocation for Parameters

```
MyClass myObj;
int x, y;
myObj = new MyClass();
x = 10;
y = 20;
y = (int)myObj.myMethod(x, y);
System.out.println(x + " " + y);
```

4

3

```
class MyClass {
  public double myMethod(int one, double y ){
      double i=5;
      one += 6;
      i *= y;
      return i;
    }
}
```



Objects and Methods

- When we pass an object, we are actually passing the reference (name) of an object
  - it means a duplicate of an object is NOT created in the called method
- The return value is also similarly copied
  - since the reference is copied, the actual object does not get destroyed!

Object Example

```
class Vector {
   int xCoord, yCoord;

   public Vector ( int x, int y) {
       xCoord = x;
       yCoord = y;
  }
   public Vector addVector(Vector v ) {
       Vector tempVector;
       tempVector  = new Vector (xCoord+v.getX(),
       yCoord + v.getY() );
       return tempVector;
   }
   public int getX() {
       return xCoord;
   }
   public int getY() {
       return yCoord;
   }
   . . .
}
```

```
Vector v1, v2;
v1 = new Vector(2,1);
v2 = new Vector(3,4);
v2 = v1.addVector(v2);
```

Memory diagram

Method Overloading

- In a given class, we can have multiple methods with the same name.
- Called overloading.
- Which one gets called?
- Based upon signature
  - Number, order, and type of parameters.
  - **NOTE**: Names of parameters and return type not included in signature!
- Overloaded methods must have unique signatures.

Encapsulation

- One of the key benefits of OOP
- Limit who can view/modify what data members and how
- Improves program reliability and reuse
- Achieved by
  - hiding data members from outside the class
  - limiting which methods can be called directly from outside the class
  - using **public** and **private** modifiers

Visibility modifiers

- A data member or method that is declared **public** can be accessed by the code in any class.
- A **private** data member can only be accessed code that is part of the same class.
- A **private** method can only be called from code that is part of the same class.

Guidelines

- Implementation details (data members) should be **private**
  - Use accessor/mutator methods
- Internal methods should be **private**
- Constructors are usually **public**
- Constants may be made **public** if useful (e.g. Math.PI)
- Default value is **public**.

Identifier types

- Identifiers can be declared almost anywhere in a program.
- There are three main types of declarations:
  - Data members of a class
    - Declared outside any method
    - Usually at the beginning of the class definition
  - Formal parameters of a method
  - Local variables inside a method

Identifier extent and scope

- Each identifier refers to a piece of memory.
- That piece is reserved upon declaration.
- The lifetime of this reservation is called the extent of the identifier.
- The ability to access this location from a given line of code is called scope.
- Important to understand both.
- Extent and scope depend upon the type of variable and its declaration.

Extent

- Object data members
  - created when an object is created (by **new**)
  - destroyed when the object is garbage collected (no more references to it)
  - must be unique within each class
- Formal parameters
  - created each time the method is called
  - destroyed when the method finishes execution
  - must be unique for each method
- Local variables
  - created upon declaration
  - destroyed at end of block
  - must be unique for each block,
- Limiting extent allows compilers to reuse space

Which one do we mean?

- An identifier in a program is matched as follows:
  - A local variable, or parameter, if it exists.
  - A data member, otherwise.
- Thus, a data member can be masked!
- Can lead to subtle errors.

PURDUE
UNIVERSITY

Sample Matching

```
class Student {

    private String    name;
    private String     id;

    public Student(String fName, String lName, String id) {

        String sName;

        sName = fName + “, “ + lName;

        name = sName;

        id  = id;

    }
    ...
}
```

Sample Matching

```
class Student {

    private String    name;
    private String     id;

    public Student(String name, String lName, String sId) {

        String sName = name + “, “ + lName;

        name = sName;

        id  = sId;

    }
    ...
}
```

Remember, ….

- A local variable can be declared just about anywhere!
- Its scope (the area of code from where it is visible) is limited to the enclosing braces.
- Statements within a pair of braces are called a block.
- Local variables are destroyed when the block finishes execution.
- Data members of a class are declared outside any method. Their scope is determined by public and private modifiers.

Reserved Word **this**

- The reserved word **this** is an automatically defined data member of each object.
- It is set to point to the object itself.
- It is called a *self-referencing pointer*

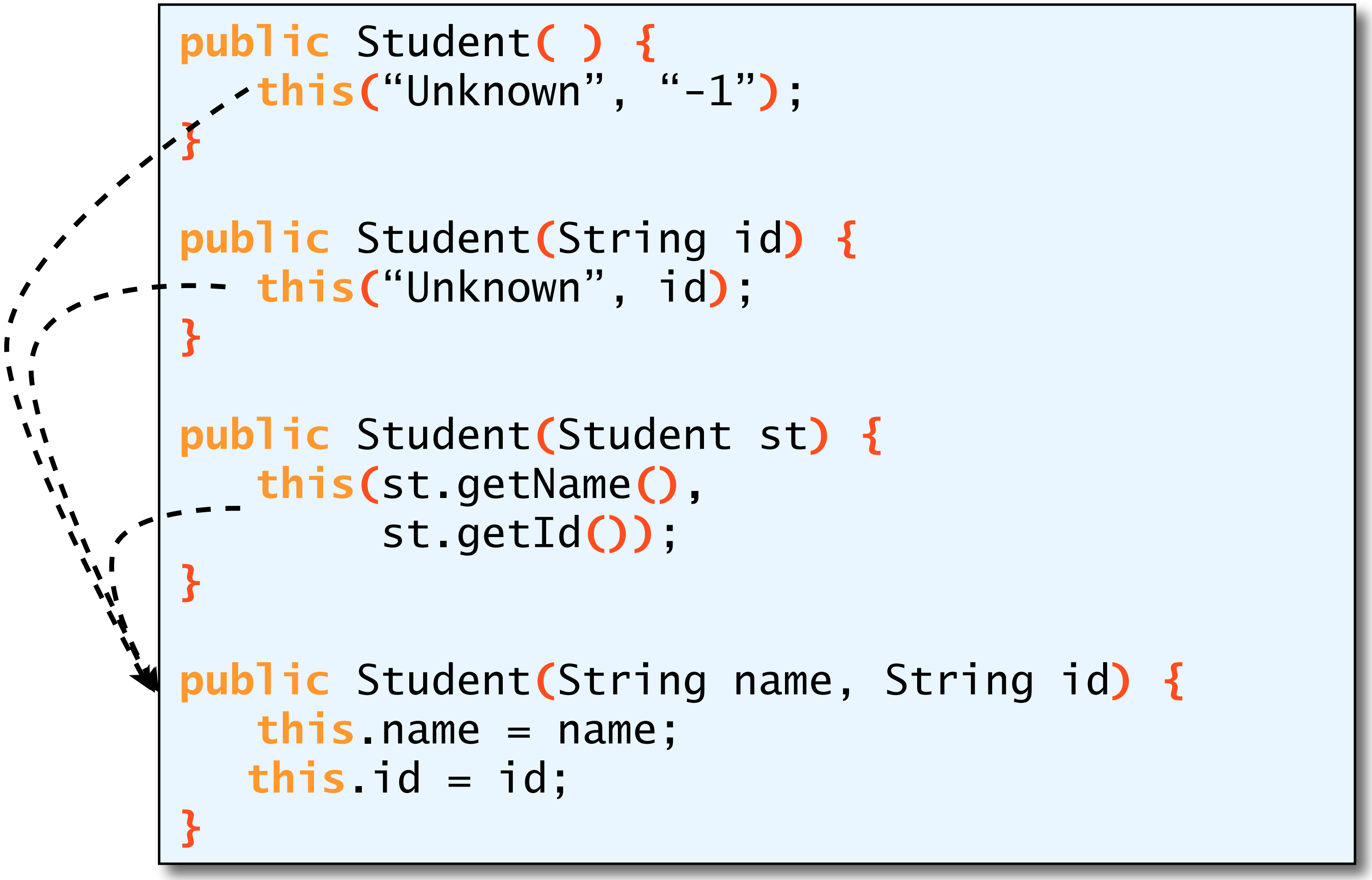
Correct references

```
class Student {

    private String    name;
    private String     id;

    public Student(String fName, String lName, String id) {

        String sName;

        sName = fName + “, “ + lName;

        name = sName;

        this.id  = id;

    }
    ...
}
```

Overloaded constructors

- As with other methods, constructors can be overloaded.
- Matching based upon signature.
- Can also call one constructor from another using the keyword **this**
  - must be the first statement in the calling constructor.

Multiple constructors and this



```
public Student( ) {
   this(“Unknown”, “-1”);
}

public Student(String id) {
   this(“Unknown”, id);
}

public Student(Student st) {
   this(st.getName(),
        st.getId());
}

public Student(String name, String id) {
   this.name = name;
   this.id = id;
}
```

PURDUE
UNIVERSITY

Copy constructor

- A copy constructor can be very handy.
- It takes an object as input and creates another object (of the same class) and copies the values.
- Useful also for preventing surreptitious access to private objects.
- If a method returns a pointer to a private object, then the client can modify the private object!
- Avoid this by returning a copy object

Use of a copy constructor

```
class Corruptor {
   Jedi luke;
   Person p;
   public static void main(String[] args){
      luke = new Jedi(new Person(“ObiWan”));
      p = luke.getFather();
      p.setName(“Darth Vader”);
      p = luke.getFather();
      System.out.println(p.getName());
   }
}
```

```
class Jedi {
   private Person father;

   public void Jedi(Person f){
      father = f;
   }

   Person getFather(){
      return father;
   }
}
```

```
class Jedi {
   private Person father;

   public void Jedi(Person f){
      father = f;
   }

   Person getFather(){
      Person x;
      x = new Person(father);
      return x;
   }
}
```

Class vs. Instance methods

- There are two main types of methods in OOP:
  - *Instance* methods that are called on an object
    - person.getAge()
    - have access to that object's data members
  - *Class* methods that do not require an object
    - Math.sqrt(), Integer.parseInt()
- Class methods are specified using the **static** modifier

Class vs. Instance methods

```
class Test {

   public static void main(String args[]) {
      myMethod();
      Person jane = new Person (“Jane”);
      jane.setAge(35);
   }
   public static void myMethod() {
      System.out.println (“Class Method”);
   }
}
```

```
class Person {
   String name;
   int age;

   public Person(String n) {
      name = n;
   }
   public void setAge(int a) {
      age = a;
   }
   public int getAge() {
      return age;
   }
}
```

Class vs. Instance Data members

- Data members too can be either
  - instance -- one copy per object, stored with object
  - class -- one copy for entire class, stored with class
- The static modifier is used to declare a class data member
- Static data members are accessed using the Class name
- Static constants can be very useful (e.g., Math.PI)

Using class variables

```
class Student {
   private static int nextID=100;
   public static final String UGRAD = “Undergraduate”;
   public static final String GRAD = “Graduate”;
   private String name;
  private String iD;
  private String status;
   public Student(String n, String stat){
      iD = “” + Student.nextID++;
      name = n;
      status = stat;
   }
}
```

```
class Test {
    public static void main(String args[]) {
        Student s1, s2;
       s1 = new Student(“Radha”, Student.UGRAD);
       s2 = new Student(“Jane”, Student.GRAD);
       System.out.println(s1.getName() + “ is an “ + s1.getStatus()
   + “ with ID:” + s1.getId());
        System.out.println(s2.getName() + “ is an “ + s2.getStatus()
      + “ with ID:” + s2.getId());
    }
}
```

Static methods

- IMPORTANT: a static method cannot access any instance data members or *instance methods*
  - I.e. it can only access other static members and methods
- Note that main is a static method!
  - No object is necessary to run main.
  - But, it can’t call non-static methods.

Class vs. Instance methods

```
class Test {

   public static void main(String args[]) {
      myClassMethod();
      Test test = new Test();
      test.myInstanceMethod();
   }
   public static void myClassMethod() {
      System.out.println (“Class Method”);
   }
   public void myInstanceMethod() {
      System.out.println (“Instance Method”);
   }
}
```

Static Initializer

- Earlier, we initialized static variables upon declaration. This initialization takes place when the class is loaded.
  - Imported or used for the first time in a program.
- What if we want to do more?
  - E.g. set the initial value based upon user input?
- We can define a static initializer segment that gets executed when a class is loaded.

Static Initializer

```
class Student {
      . . .
   private static int nextID;
   . . .
   static {
           String str;
           str = JOptionPane.showInputDialog(null, “enter starting
value”);
           nextID = Integer.parseInt(str);
   }
   . . .
```

- As with static methods, we cannot reference any non-static method or data member from the static initializer block.

Examples of class methods

- The Math class has numerous class methods and constants
  - `Math.abs, Math.pow,`
  - `Math.PI`
- We have also seen Wrapper classes for the primitive data types:
  - `Integer: Integer.parseInt, Integer.MAX_VALUE`
  - `Double: Double.parseDouble,` …
  - Similarly for **`long`**, **`short`**, **`byte`**, and **`boolean`**.

Changing Any Class to a Main Class

- Any class can be set to be a main class.
- All you have to do is to include the main method.

```
class Student {

    . . .
    public static void main(String[] args) {

        Student student1;

        student1 = new Student( );
       student1.setName("Purdue Pete");

        System.out.println(student1.getName() + "is a 
student");
    }
}
```

- It can be executed by: %`java Student`

The null constant

- null is a special value. Its type is that of a reference to an object (of any class).
- We can set an object identifier to this value to show that it does not point to any object.
  - Bicycle bike1=null;
- A method that returns objects (of any class) can return a null value.
- Note that you will get a run-time error if you access a data member of call a method of a null object -- *null pointer exception*.

Testing for null values.

```
class Account {
   private Person owner;
   public Account(){
      owner=null;
   }
   public void setOwner(Person p){
      owner = p;
   }
   public Person getOwner(){
      return(owner);
   }
}
```

We can use == or != to check if an object reference is null or not.

```
class Bank {
public static void main(String[] arg){
      Account acc = new Account();
      Person p;
        …
      p = acc.getOwner();
      if (p==null)
        System.out.println(“No owner”);
      …
}
```