Object-Oriented Programming

- Programming defined in terms:

  1) objects (nouns) and

  2) relationships between objects

  Object-Oriented programming languages:

  1) SmallTalk

  2) C++

  3) C#

  4) Java

What is a Class?

- A class is a blueprint that defines the variables and methods common to all objects of a certain kind.
- Example: ‘your dog’ is a object of the class Dog.
- An object holds values for the variables defines in the class.
- An object is called an instance of the Class

Objects

- Everything in Java is an object.

    Well ... almost.

  Object lifecycle:

  1) creation

  2) usage

  3) destruction

Class

- A basis for the Java language.
- Each concept we wish to describe in Java must be included inside a class.
- A class defines a new data type, whose values are objects:

  1) a class is a template for objects

  2) an object is an instance of a class

Class Definition

- A class contains a name, several variable declarations (instance variables) and several method declarations. All are called members of the class.
- General form of a class:
- [public] [(abstract | final)]class classname [extends superclass] [implements interface]{
- type instance-variable-1;
- …
- type instance-variable-n;
- type method-name-1(parameter-list) { … }
- type method-name-2(parameter-list) { … }
- …
- type method-name-m(parameter-list) { … }
- }

Class Definition

1. Modifiers :A class can have public or default (no modifier) visibility.
   It can be either abstract, final or concrete(no modifier).

2. Class name : It must have the class keyword, and class must be followed by a legal identifier.
3. Superclass(if any):It may optionally extend one parent class. By default, it will extend java.lang.Object
4. Interfaces(if any):It may optionally implement any number of comma-separated interfaces.
5. Body: The class's variables and methods are declared within a set of curly braces '{}'.

Example: Class

- A class with three variable members:

-
  ```
  class Box {
          double width;
          double height;
          double depth;
  }
  ```
- A new Box object is created and a new value assigned to its width variable:

  ```
  Box myBox = new Box();
  myBox.width = 100;
  ```

Object

- It is a basic unit of Object Oriented Programming and represents the real life entities.
- A typical Java program creates many objects, which as you know, interact by invoking methods.
- An object consists of :

1. State : It is represented by attributes of an object. It also reflects the properties of an object.
2. Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. Identity : It gives a unique name to an object and enables one object to interact with other objects.

- Example of an object : Car

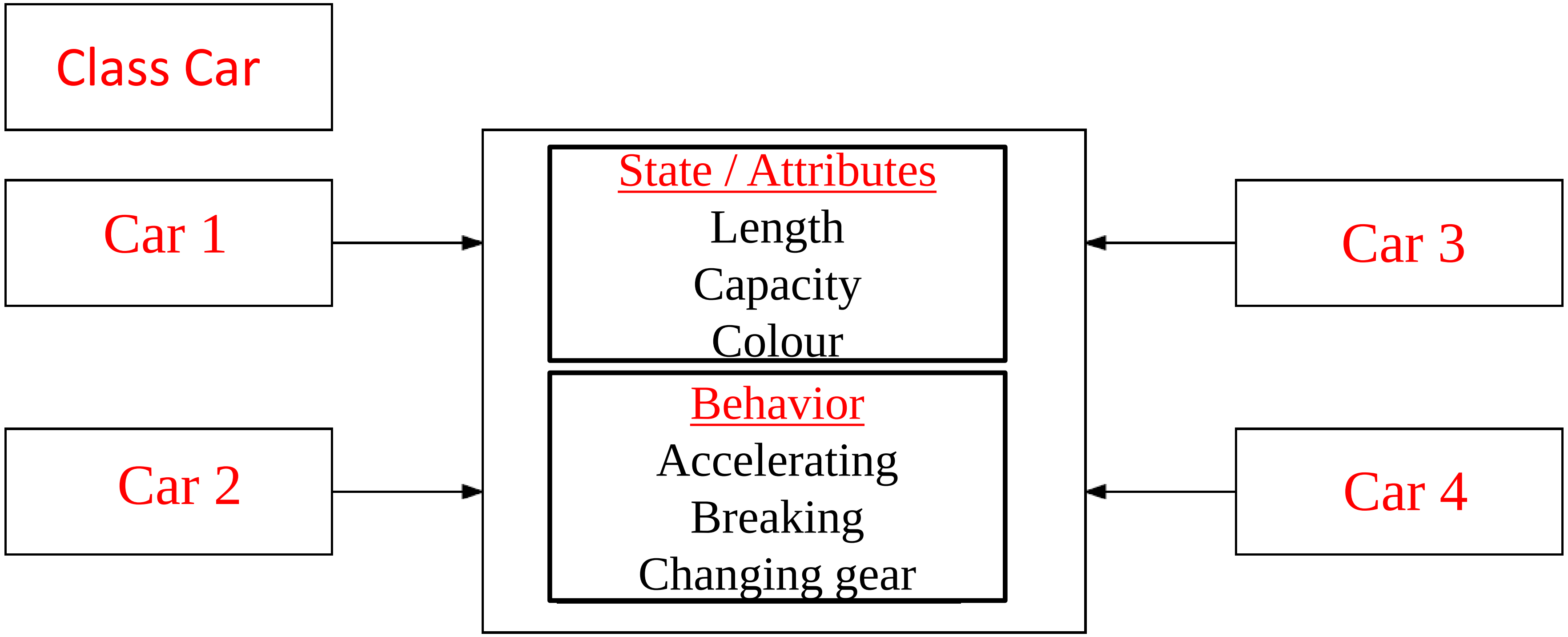
Identity
Name of the car

State / Attributes
Length
Capacity
Colour

Behavior
Accelerating
Breaking
Changing gear

Object

- When an object of a class is created, the class is said to be **instantiated**.
- All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.



Declaring Objects

- Obtaining objects of a class is a two-stage process:

  1) Declare a variable of the class type:

  ```
  Box myBox;
  ```

- The value of myBox is a reference to an object, if one exists, or null.
- At this moment, the value of myBox is null.

  2) Acquire an actual, physical copy of an object and assign its address to the variable. How to do this?

Operator new

- Allocates memory for a Box object and returns its address:

```
Box myBox = new Box();
```

- The address is then stored in the myBox reference variable.

- Box() is a class constructor - a class may declare its own constructor or rely on the default constructor provided by the Java environment.

Memory Allocation

- Memory is allocated for objects dynamically.
- This has both advantages and disadvantages:

  1) as many objects are created as needed

  2) allocation is uncertain – memory may be insufficient
- Variables of simple types do not require new:

```
int n = 1;
```

- In the interest of efficiency, Java does not implement simple types as objects. Variables of simple types hold values, not references.

Assigning Reference Variables

- Assignment copies address, not the actual value:

```
Box b1 = new Box();
Box b2 = b1;
```

- Both variables point to the same object.
- Variables are not in any way connected. After

```
b1 = null;
```

- b2 still refers to the original object.

Variable Independence

- Each object has its own copy of the instance variables: changing the variables of one object has no effect on the variables of another object.
- Consider this example:
- class BoxDemo2 {
- public static void main(String args[]) {

```
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
```

Variable Independence

```
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
        }
}
```

- What are the printed volumes of both boxes?

Differences between Objects and Classes

Objects	**Class**
Objects is an instance of a class	Class is a blueprint from which objects are created
Object is a real world entity such as pen, pencil, mobile etc.	Class is a group of similar objects
Object is a physical entity.	Class is a logical entity
Object is created many times as per requirement.	Class is declared once
Object allocates memory when it is created.	Class does not allocates memory when it is created.
Object are created through new keyword. Student st = new Student();	Class is declared using class keyword. class Student{ }

Methods

- A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation and they are also known as functions.
- It is used to achieve the reusability of code.
- We write a method once and use it many times. We do not require to write code again and again.
- It also provides the easy modification and readability of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.
- A method is a block of code which only runs when it is called.

Types of Java methods

- Depending on whether a method is defined by the user, or available in the standard library, there are two types of methods in Java:

1. Standard Library Methods
2. User-defined Methods

- Standard Library Methods - The standard library methods are built-in methods in Java that are readily available for use.
- These standard libraries come along with the Java Class Library (JCL) in a Java archive (*.jar) file with JVM and JRE. For example,
- print() is a method of java.io.PrintSteam.
- sqrt() is a method of Math class. It returns the square root of a number.

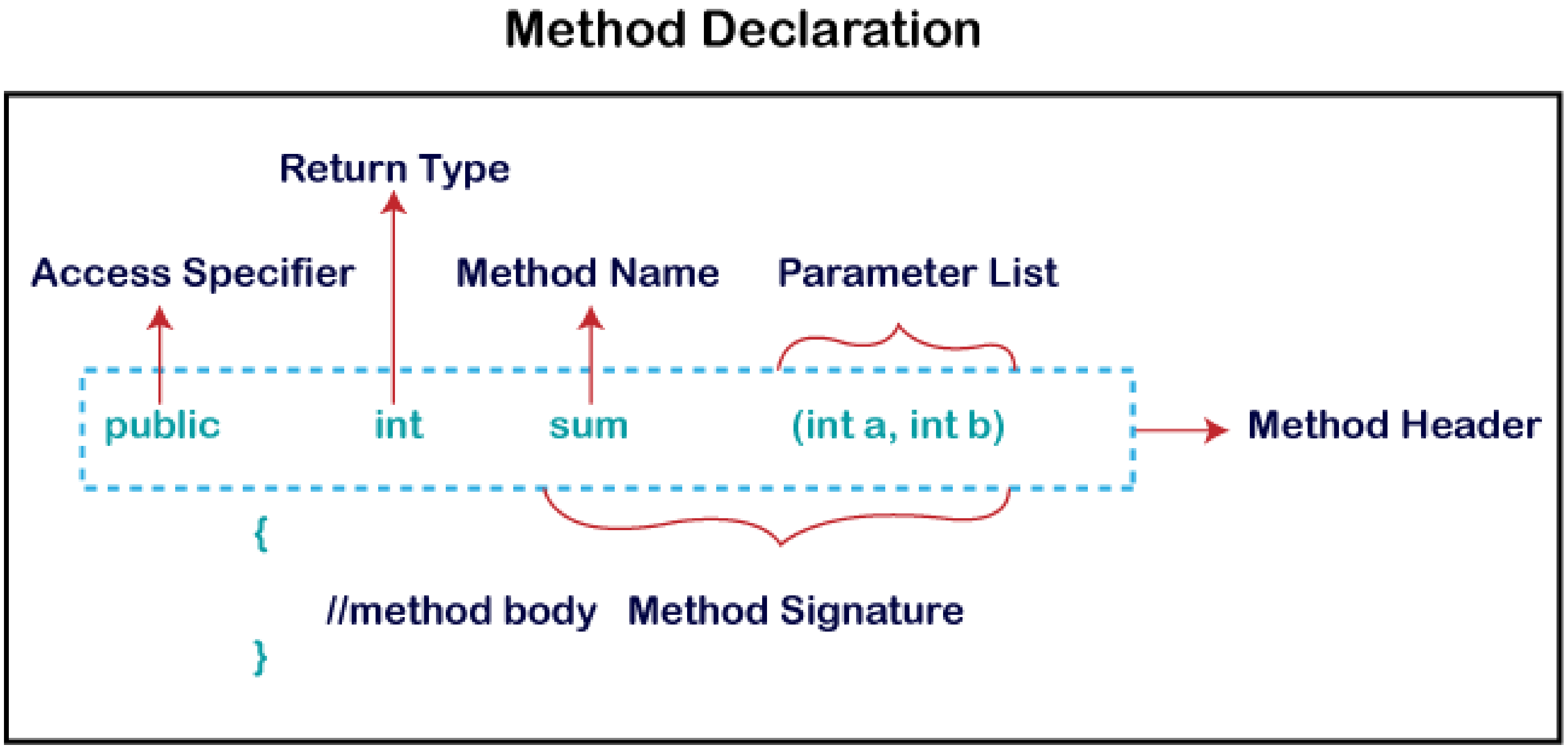
User Defined Methods

- General form of a method definition:

```
type name(parameter-list) {
        … return value; …
        }
```

- Components:
- 1) type - type of values returned by the method. If a method does not return any value, its return type must be void.
- 2) name is the name of the method
- 3) parameter-list is a sequence of type-identifier lists separated by commas
- 4) return value indicates what value is returned by the method.

Example: Method



Example: Method

- Classes declare methods to hide their internal data structures, as well as for their own internal use:
- Within a class, we can refer directly to its member variables:
- class Box {

```
    double width, height, depth;
    void volume() {
        System.out.print("Volume is ");
        System.out.println(width * height * depth);
        }
    }
```

Example: Method 2

- When an instance variable is accessed by code that is not part of the class in which that variable is defined, access must be done through an object:
- class BoxDemo3 {

```
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        mybox1.width = 10; mybox2.width = 3;
        mybox1.height = 20; mybox2.height = 6;
        mybox1.depth = 15; mybox2.depth = 9;
        mybox1.volume();
        mybox2.volume();
        }
}
```

Value-Returning Method

- The type of an expression returning value from a method must agree with the return type of this method:
- class Box {

```
        double width;
        double height;
        double depth;
        double volume() {
                return width * height * depth;
                }
        }
```

Value-Returning Method

- class BoxDemo4 {

```
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        mybox1.width = 10;
        mybox2.width = 3;
        mybox1.height = 20;
        mybox2.height = 6;
        mybox1.depth = 15;
        mybox2.depth = 9;
```

Value-Returning Method

- The type of a variable assigned the value returned by a method must agree with the return type of this method:

```
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

Parameterized Method

- Parameters increase generality and applicability of a method:

  1) method without parameters

  ```
  int square() { return 10*10; }
  ```

  2) method with parameters

  ```
  int square(int i) { return i*i; }
  ```

- **Parameter**: a variable receiving value at the time the method is invoked.
- **Argument**: a value passed to the method when it is invoked.

Example: Parameterized Method

-
  ```
  class Box {
          double width;
          double height;
          double depth;
          double volume() {
                  return width * height * depth;
          }
          void setDim(double w, double h, double d) {
                  width = w; height = h; depth = d;
          }
  }
  ```

Example: Parameterized Method

```
•class BoxDemo5 {
        public static void main(String args[]) {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
                mybox1.setDim(10, 20, 15);
                mybox2.setDim(3, 6, 9);
                vol = mybox1.volume();
                        System.out.println("Volume is " + vol);
                vol = mybox2.volume();
                        System.out.println("Volume is " + vol);
        }
•}
```

Constructor

- A constructor initializes the instance variables of an object.
- It is called immediately after the object is created but before the new operator completes.

  1) it is syntactically similar to a method:

  2) it has the same name as the name of its class

  3) it is written without return type; the default return type of a class constructor is the same class
- When the class has no constructor, the default constructor automatically initializes all its instance variables with zero.

Example: Constructor

- class Box {

```
        double width;
        double height;
        double depth;
        Box() {
                System.out.println("Constructing Box");
                width = 10; height = 10; depth = 10;
                }
        double volume() {
                return width * height * depth;
                }
        }
```

Example: Constructor

```
class BoxDemo6 {
    public static void main(String args[]) {
    Box mybox1 = new Box();
    Box mybox2 = new Box();
    double vol;
    vol = mybox1.volume();
    System.out.println("Volume is " + vol);
    vol = mybox2.volume();
    System.out.println("Volume is " + vol);
    }
}
```

Parameterized Constructor

- So far, all boxes have the same dimensions.
- We need a constructor able to create boxes with different dimensions:

```
class Box {
        double width;
        double height;
        double depth;
        Box (double w, double h, double d) {
                width = w; height = h; depth = d;
                }
double volume() {
                return width * height * depth; }
}
```

Parameterized Constructor

```
• class BoxDemo7 {
        public static void main(String args[]) {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
  }
```

Difference between Constructor and Method

1. The purpose of constructor is to create object of a class while the purpose of a method is to perform a task by executing java code.
2. Constructors cannot be abstract, final, static and synchronised while methods can be.
3. Constructors do not have return types while methods do.

Static Members

- Static keyword can be used with class, variable, method and block.
- Variables and methods declared using keyword static are called static members of a class.
- The non-static variables and methods belong to instance.
- The static members (variables, methods) belong to class. Static members are not part of any instance of the class.
- Static members can be accessed using class name directly, in other words, there is no need to create instance of the class specifically to use them.

Static Variables

- Static variables are also called class variables because they can be accessed using class name.
- Static variable in Java is variable which belongs to the class and initialized only once at the start of the execution. It is a variable which belongs to the class and not to object(instance).
- Non static variables are called instance variables and can be accessed using instance reference only.
- Static variables occupy single location in the memory. These can also be accessed using instance reference.
- Static variables are initialized only once, at the start of the execution. These variables will be initialized first, before the initialization of any instance variables.

Static Variables

```
class Test {
   // static variable
   static int max = 10;
     // non-static variable
   int min = 5;
}
public class StaticVar {
   public static void main(String[] args) {
       Test obj = new Test();
       // access the non-static variable
       System.out.println("min + 1 = " + (obj.min + 1));
       // access the static variable
       System.out.println("max + 1 = " + (Test.max + 1));
   }
}
```

Static Methods/Class Methods

- Static methods are also called class methods. A static method belongs to class, it can be used with class name directly. It is also accessible using instance references.
- Static methods can use static variables only, whereas non-static methods can use both instance variables and static variables.
- A static method can call only other static methods and can not call a non-static method from it.
- A static method can be accessed directly by the class name and doesn’t need any object
- A static method cannot refer to "this" or "super" keywords in anyway.
- Note: main method is static, since it must be accessible for an application to run, before any instantiation takes place.

Static Methods/Class Methods

```
class StaticTest {
    // non-static method
    int multiply(int a, int b){
        return a * b;
    }
    // static method
         static int add(int a, int b){
        return a + b;
    }
 }
      public class Main {
          public static void main( String[] args ) {
         StaticTest st = new StaticTest();
        // call the nonstatic method
        System.out.println(" 2 * 2 = " + st.multiply(2,2));
        // call the static method
        System.out.println(" 2 + 3 = " + StaticTest.add(2,3));
   }
}
```

finalize() Method

- A constructor helps to initialize an object just after it has been created.
- In contrast, the finalize method is invoked just before the object is destroyed:

  1) implemented inside a class as:

  ```
  protected void finalize() { … }
  ```

  2) implemented when the usual way of removing objects from memory is insufficient, and some special actions has to be carried out
- How is the finalize method invoked?

finalize() Method

- To add a finalizer to a class, you simply define the finalize( ) method.
- The Java run time calls that method whenever it is about to recycle an object of that class.
- Inside the finalize( ) method you will specify those actions that must be performed before an object is destroyed.
- The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects.
- Right before an asset is freed, the Java run time calls the finalize( ) method on the object

Garbage Collection

- Garbage collection is a mechanism to remove objects from memory when they are no longer needed.
- Garbage collection is carried out by the garbage collector:

1. The garbage collector keeps track of how many references an object has.
2. It removes an object from memory when it has no longer any references.
3. Thereafter, the memory occupied by the object can be allocated again.
4. The garbage collector invokes the finalize method.

Keyword this

- Keyword this allows a method to refer to the object that invoked it.
- It can be used inside any method to refer to the current object:
- Box(double width, double height, double depth) {

```
        this.width = width;
        this.height = height;
        this.depth = depth;
    }
```

- The above use of this is redundant but correct.
- When is this really needed?

Instance Variable Hiding

- Variables with the same names:
  1. it is illegal to declare two local variables with the same name inside the same or enclosing scopes
  2. it is legal to declare local variables or parameters with the same name as the instance variables of the class.
- As the same-named local variables/parameters will hide the instance variables, using this is necessary to regain access to them:

```
Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
}
```

Method Overloading

- It is legal for a class to have two or more methods with the same name.
- However, Java has to be able to uniquely associate the invocation of a method with its definition relying on the number and types of arguments.
- Therefore the same-named methods must be distinguished:

    1) by the number of arguments, or

    2) by the types of arguments
- Overloading and inheritance are two ways to implement polymorphism.

Example: Overloading 1

-
  ```
  class OverloadDemo {
      void test() {
          System.out.println("No parameters");
      }
      void test(int a) {
          System.out.println("a: " + a);
      }
      void test(int a, int b) {
          System.out.println("a and b: " + a + " " + b);
      }
      double test(double a) {
          System.out.println("double a: " + a); return a*a;
      }
  }
  ```

Example: Overloading 2

```
• class Overload {
        public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
                double result;
                ob.test();
                ob.test(10);
                ob.test(10, 20);
                result = ob.test(123.2);
                System.out.println("ob.test(123.2): " + result);
        }
}
```

Different Result Types

- Different result types are insufficient.
- The following will not compile:
-
```
double test(double a) {
        System.out.println("double a: " + a);
        return a*a;
        }
        int test(double a) {
                System.out.println("double a: " + a);
                return (int) a*a;
        }
```

Overloading and Conversion 1

- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method’s parameters.
- When no exact match can be found, Java’s automatic type conversion can aid overload resolution:
- class OverloadDemo {

```
void test() {
        System.out.println("No parameters");
        }
void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
}
```

Overloading and Conversion

-
  ```
  void test(double a) {
                  System.out.println("Inside test(double) a: " + a);
      }
  }
  class Overload {
      public static void main(String args[]) {
      OverloadDemo ob = new OverloadDemo();
      int i = 88;
      ob.test();
      ob.test(10, 20);
      ob.test(i);
      ob.test(123.2);
      }
  }
  ```

Constructor Overloading

- Why overload constructors? Consider this:
- class Box {

```
	double width, height, depth;
	Box(double w, double h, double d) {
		width = w; height = h; depth = d;
	}
	double volume() {
		return width * height * depth;
	}
}
```

- All Box objects can be created in one way: passing all three dimensions.

Constructor Overloading

- Three constructors: 3-parameter, 1-parameter, parameter-less.

```
class Box {
      double width, height, depth;
      Box(double w, double h, double d) {
            width = w; height = h; depth = d;
            }
      Box() {
            width = -1; height = -1; depth = -1;
            }
      Box(double len) {
            width = height = depth = len;
            }
            double volume() { return width * height * depth; }
      }
```

Example: Overloading 2

```
class OverloadCons {
	public static void main(String args[]) {
	Box mybox1 = new Box(10, 20, 15);
	Box mybox2 = new Box();
	Box mycube = new Box(7);
	double vol;
	vol = mybox1.volume();
	System.out.println("Volume of mybox1 is " + vol);
	vol = mybox2.volume();
	System.out.println("Volume of mybox2 is " + vol);
	vol = mycube.volume();
	System.out.println("Volume of mycube is " + vol);
	}
}
```

Using Objects as Parameters

```
• class Box {
        double width;
       double height;
       double depth;
// construct clone of an object
 Box(Box ob) { // pass object to constructor
          width = ob.width;
          height = ob.height;
          depth = ob.depth;
   }
```

Using Objects as Parameters

```
•  Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
   }
   double volume() {
        return width * height * depth;
      }
• }
•   class ObjectCons {
    public static void main(String args[]) {
// create boxes using the various constructors
            double vol;
            Box mybox1 = new Box(10, 20, 15);
    // get volume of first box
    vol = mybox1.volume();
            System.out.println("Volume of mybox1 is " + vol);
            Box myclone = new Box(mybox1);
//get volume of clone
            vol = myclone.volume();
            System.out.println("Volume of clone is " + vol);
             }
•   }
```

Argument Passing

- There are two ways that a computer language can pass an argument to a subroutine.
- The first way is ***call-by-value***. *This method copies the value of an argument* into the formal parameter of the subroutine.
- Therefore, changes made to the parameter of the subroutine have no effect on the argument.
- The second way an argument can be passed is ***call-by-reference.***
- *In this method, a reference to an argument (not the value of*

  the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

call-by-value

- // Simple Types are passed by value.
- class Test {

```
        void meth(int i, int j) {
            i *= 2;
            j /= 2;
        }
```

- }

call-by-value

- class CallByValue {
-   public static void main(String args[]) {

        Test ob = new Test();

          int a = 15, b = 20;

  System.out.println("a and b before call: " + a + " " + b);

      ob.meth(a, b);

    System.out.println("a and b after call: " + a + " " + b);

      }
- }

call-by-reference.

```
• class Test {
          int a, b;
          Test (int i, int j) {
                  a = i;  b = j;
          }
// pass an object
          void meth(Test o) {
                  o.a *=  2;   o.b /= 2;
          }
• }
```

call-by-reference.

- class CallByRef {
  public static void main(String args[]) {
          Test ob = new Test(15, 20);
  System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
       ob.meth(ob);

        System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
-   }
- }

Inheritance

- One of the pillars of object-orientation.
- A new class is derived from an existing class:
  1) existing class is called super-class
  2) derived class is called sub-class
- A sub-class is a specialized version of its super-class:
  1) has all non-private members of its super-class
  2) may provide its own implementation of super class methods
- Objects of a sub-class are a special kind of objects of a super-class.

Inheritance

- **Inheritance** is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.
- Important terminology:
- Super Class**:** The class whose features are inherited is known as super class(or a base class or a parent class).
- Sub Class: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class).
- The subclass can add its own fields and methods in addition to the superclass fields and methods.
- Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class.
- By doing this, we are reusing the fields and methods of the existing class.

Inheritance Syntax

- Syntax:
- class Super-class {
-     .....
-     .....
- }
- class Sub-class extends Super-class {
- …
- }
- Each class has at most one super-class; no multi-inheritance in Java.
- No class is a sub-class of itself.

Example: Super-Class

-
  ```
  class A {
          int i;
          void showi() {
                  System.out.println("i: " + i);
                  }
          }
  ```

Example: Sub-Class

- class B extends A {
```
        int j;
        void showj() {
                System.out.println(“j: " + j);
        }
        void sum() {
                System.out.println("i+j: " + (i+j));
        }
}
```

Example: Testing Class

- class SimpleInheritance {

```
        public static void main(String args[]) {
                A  ob1= new A();
                B ob2 = new B();
                 ob1.i = 10;
                System.out.println("Contents of a: ");
                 ob1.showi();
                 ob2.i = 7; ob2.j = 8;
                System.out.println("Contents of b: ");
                 ob2.showi(); ob2.showj();
                System.out.println("Sum of I and j in b:");
                 ob2.sum();
                }
```

- }

Inheritance and Private Members

- A class may declare some of its members private.
- A sub-class has no access to the private members of its super-class:

-
  ```
  class A {
        int i;
        private int j;
        void setij(int x, int y) {
              i = x; j = y;
        }
  }
  ```

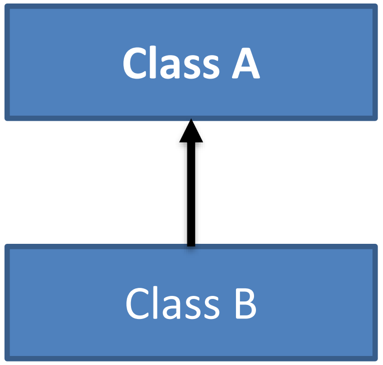
Inheritance and Private Members

- Class B has no access to the A's private variable j.
- This program will not compile:

```
class B extends A {
        int total;
        void sum() {
                total = i + j;
        }
}
```

Inheritance

- Single Inheritance
- In single inheritance, one class inherits the properties of another. It enables a derived class to inherit the properties and behavior from a single parent class. This will, in turn, enable code reusability as well as add new features to the existing code.
- Class A is your parent class and Class B is your child class which inherits the properties and behavior of the parent class.



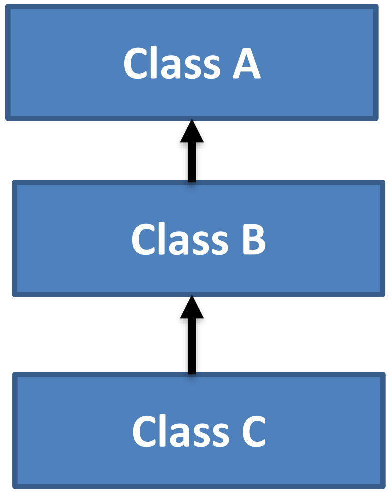
Inheritance

- **Single Inheritance**
- class Animal{

```
        void eat(){
              System.out.println(“eating”);
        }
   }
   class Dog extends Animal{
        void bark(){
              System.out.println(“barking”);
        }
   }
class TestInheritance{
   public static void main(String args[]){
              Dog d=new Dog();
              d.bark();
              d.eat();
}
```

Inheritance

- **Multi-level Inheritance**
- When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent class but at different levels, such type of inheritance is called Multilevel Inheritance.
- class B inherits the properties and behavior of class A and class C inherits the properties of class B.
- Here A is the parent class for B and class B is the parent class for C. So in this case class C implicitly inherits the properties and methods of class A along with Class B. That's what is multilevel inheritance.



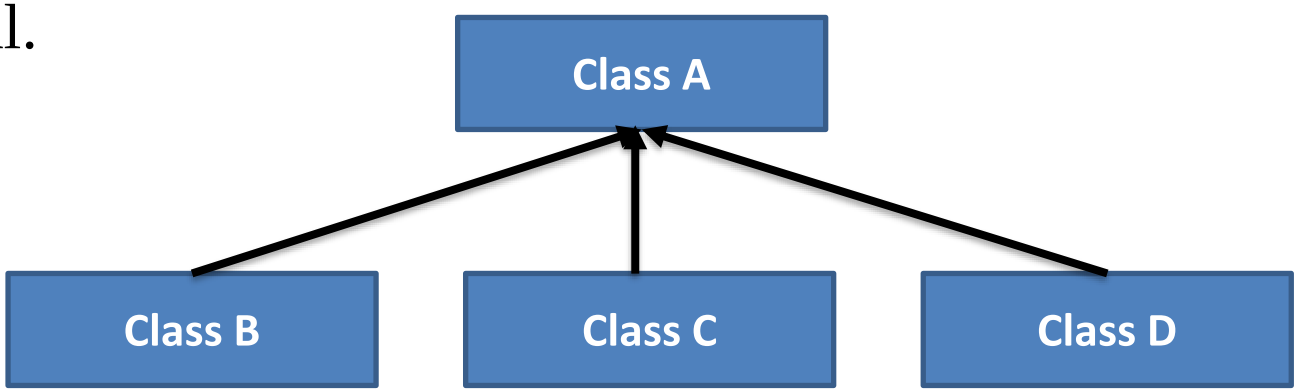
Inheritance

Multi-level Inheritance

```
classAnimal{
 voideat(){
   System.out.println(“eating…”);
 }
}
class Dog extends Animal{
 void bark(){
     System.out.println(“barking…”);
 }
}
class Puppy extends Dog{
  void weep() {
   System.out.println(“weeping…”);
  }
}
class TestInheritance2{
  public static void main(String args[]){
            Puppy d=new Puppy();
            d.weep();
            d.bark();
            d.eat();
   }
}
```

Inheritance

- Hierarchical Inheritance
- When a class has more than one child classes (subclasses) or in other words, more than one child classes have the same parent class, then such kind of inheritance is known as hierarchical.



- In the above flowchart, Class B ,class C and class D are the child classes which are inheriting from the parent class i.e Class A.

Inheritance

- Hierarchical Inheritance

```
class Animal{
    void eat(){
        System.out.println("eating…");
    }
}
class Dog extends Animal{
    void bark(){
        System.out.println("barking…");
    }
}
class Cat extends Animal{
    void meow(){
        System.out.println("meowing..");
    }
}
class TestInheritance3{
  public static void main(String args[]){
        Cat c=new Cat();
        c.meow();
        c.eat();
    }
}
```

Super

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- It is used to call superclass methods, and to access the superclass constructor.
- The most common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name.
- To understand the super keyword, you should have a basic understanding of Inheritance and Polymorphism.

Uses of Super

- Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword super.

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Super – using with variable

- super is used to refer immediate parent class instance variable.
- We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.
- If a class has a variable with name same as the variable of the superclass then, you can differentiate the superclass variable from the subclass variable using the super keyword.
- general form:
- super.member
- Here, member can be either a method or an instance variable.

Super – using with variable

```
• class Computer{
        String color="white";
  }
  class Dell extends Computer{
        String color="black";
        void displayColor(){
                System.out.println(color);          //prints color of Dell class
                System.out.println(super.color);//color of computer class
        }
• }
  class TestSuper1{
  public static void main(String args[]){
        Dell d=new Dell();
        d.displayColor();
    }
• }
```

Super – using with methods

- This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword.
- class Base{
-       void message(){
-         System.out.println(“Now at base class method ");
-       }
- }

Super – using with methods

- class Child extends Base{

        void message(){

            super.message();

            System.out.println(“Now at child class method");

      }
- }

  class SuperMethod{

        public static void main(String args[]){

                Child s=new Child();
-                 s.message();
-        }
- }

Super as a Constructor

- The super keyword can also be used to access the parent class constructor by adding ‘()’ after it, i.e. super().
- One more important thing is that ‘super()’ can call both parametric as well as non-parametric constructors depending upon the situation.
- Calling a constructor of a super-class from the constructor of a sub-class:

```
super(parameter-list);
```

- Must occur as the very first instructor in the sub-class constructor:

```
class SuperClass { … }
      class SubClass extends SuperClass {
                SubClass(…) {
                super(…);
                }
      …
}
```

Example: Super Constructor

```
•   class Computer{
•           Computer(){
                  System.out.println(“Computer class Constructor");
             }
    }
•       class Dell extends Computer{
             Dell(){
                  super();
                  System.out.println(“Dell class Constructor");
             }
•       }
      class TestSuper1{
           public static void main(String args[]){
•                 Dell d=new Dell();
•       }
•   }
```

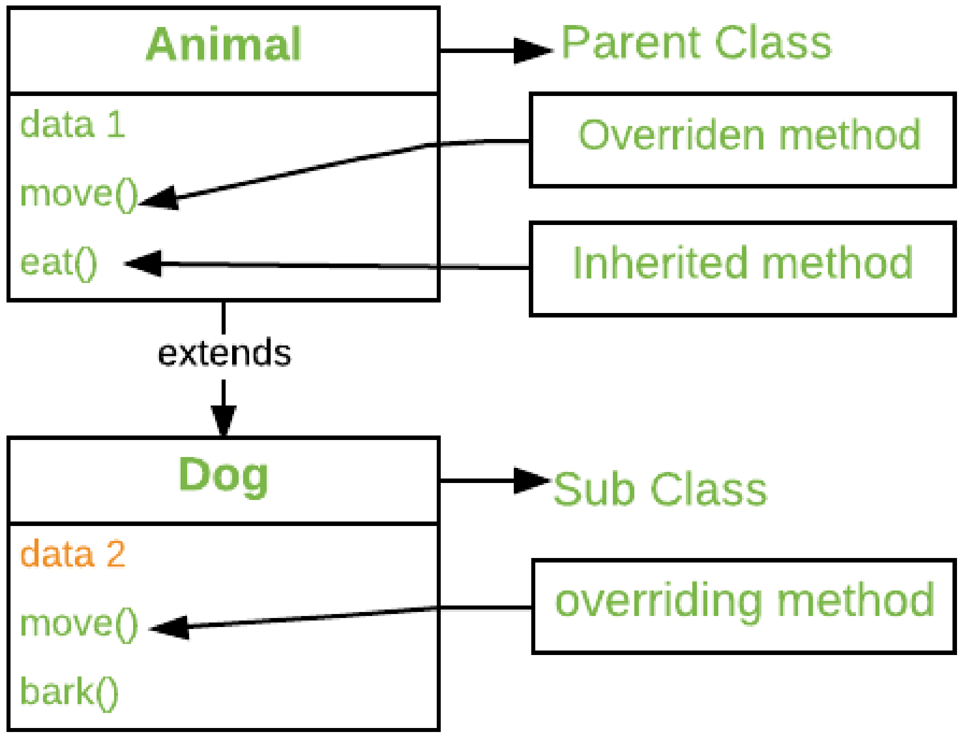
Difference between super and super()

Super	Super()
The super keyword in Java is a reference variable that is used to refer parent class objects.	The super() in Java is a reference variable that is used to refer parent class constructors.
super can be used to call parent class' variables and methods.	super() can be used to call parent class' constructors only.
The variables and methods to be called through super keyword can be done at any time,	Call to super() must be first statement in Derived(Student) Class constructor.
If one does not explicitly invoke a superclass variables or methods, by using super keyword, then nothing happens	If a constructor does not explicitly invoke a superclass constructor by using super(), the Java compiler automatically inserts a call to the no-argument constructor of the superclass.

Method Overriding

- Declaring a method in sub class which is already present in parent class is known as method overriding.
- Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class.
- In this case the method in parent class is called overridden method and the method in child class is called overriding method.
- Method overriding is one of the way by which java achieve Run Time Polymorphism.

Method Overriding



Method Overriding

- we call a method as overriding if it shares these features with one of its superclass' method:

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).
4. he same or covariant return type

Example: Hiding with Overriding

-
  ```
  class A {
      int i, j;
      A(int a, int b) {
          i = a; j = b;
      }
      void show() {
          System.out.println("i and j: " + i + " " + j);
      }
  }
  ```

Example: Hiding with Overriding

-
  ```
  class B extends A {
          int k;
          B(int a, int b, int c) {
          super(a, b);
                  k = c;
          }
          void show() {
                  System.out.println("k: " + k);
          }
  }
  ```

Example: Hiding with Overriding

- When show() is invoked on an object of type B, the version of show()
- defined in B is used:
- class Override {

  ```
  public static void main(String args[]) {
          B subOb = new B(1, 2, 3);
          subOb.show();
  }
  ```
- }
- The version of show() in A is hidden through overriding.

Rules For Method Overriding

- The access modifier can only allow more access for the overridden method.
- A final method does not support method overriding.
- A static method cannot be overridden.
- Private methods cannot be overridden.
- The return type of the overriding method must be the same.
- We can call the parent class method in the overriding method using the super keyword.
- A constructor cannot be overridden because a child class and a parent class cannot have the constructor with the same name.

Overriding versus Overloading

- The show() method in B takes a String parameter, while the show() method in A takes no parameters:
- class B extends A {

```
        int k;
        B(int a, int b, int c) {
                super(a, b); k = c;
        }
        void show(String msg) {
                System.out.println(msg + k);
        }
```

- }

Super and Method Overriding

- The hidden super-class method may be invoked using super:
-
```
class B extends A {
        int k;
        B(int a, int b, int c) {
                super(a, b);
        k = c;
    }
        void show() {
                super.show();
                System.out.println("k: " + k);
        }
    }
```
- The super-class version of show() is called within the sub-class’s version.

Overriding versus Overloading

- Method overriding occurs only when the names and types of the two methods (super-class and sub-class methods) are identical.
- If not identical, the two methods are simply overloaded:

```
class A {
        int i, j;
        A(int a, int b) {
                i = a; j = b;
        }
void show() {
        System.out.println("i and j: " + i + " " + j);
        }
}
```

Overriding versus Overloading

- The two invocations of show() are resolved through the number of arguments (zero versus one):

```
• class Override {
      public static void main(String args[]) {
          B subOb = new B(1, 2, 3);
          subOb.show("This is k: ");
          subOb.show();
      }
• }
```

Overriding versus Overloading

Overloading	Overriding
1. Whenever same method or Constructor is existing multiple times within a class either with different number of parameter or with different type of parameter or with different order of parameter is known as Overloading.	Whenever same method name is existing multiple time in both base and derived class with same number of parameter or same type of parameter or same order of parameters is known as Overriding.
2. Arguments of method must be different at least arguments.	Argument of method must be same including order.

Overriding versus Overloading

Overloading	Overriding
3. Method signature must be different.	Method signature must be same.
4. Private, static and final methods can be overloaded.	Private, static and final methods can not be override.
5. Access modifiers point of view no restriction.	Access modifiers point of view not reduced scope of Access modifiers but increased.

Overriding versus Overloading

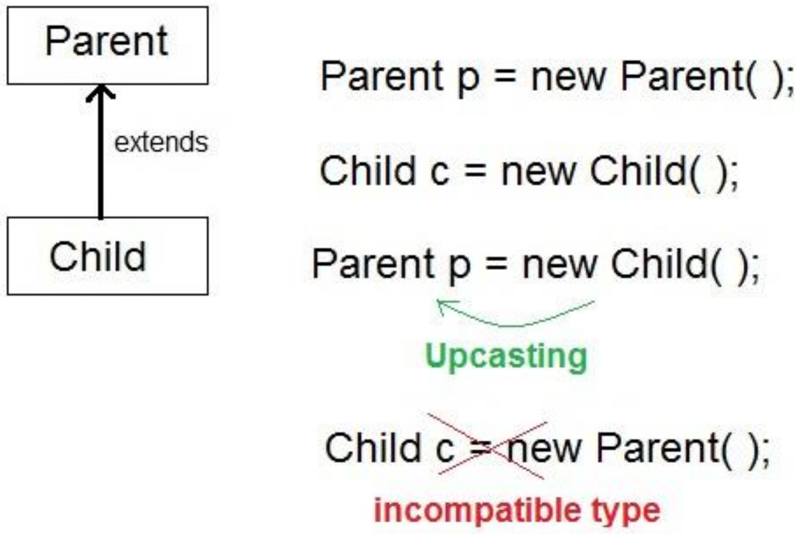
Overloading	Overriding
6. Also known as compile time polymorphism or static polymorphism or early binding.	Also known as run time polymorphism or dynamic polymorphism or late binding.
7. Overloading can be exhibited both are method and constructor level.	Overriding can be exhibited only at method label.
8. The scope of overloading is within the class.	The scope of Overriding is base class and derived class.
9. Overloading can be done at both static and non-static methods	Overriding can be done only at non-static method.

Dynamic Method Invocation

- Method overriding is one of the ways in which Java supports Runtime Polymorphism.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
- When an overridden method is called through a superclass reference, Java determines which version (superclass / subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs.
- Thus, this determination is made at run time.

Dynamic Method Invocation

- A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden methods at run time.
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.



```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );

Child c = new Parent( );
```

Dynamic Method Invocation

- Overriding is a lot more than the namespace convention.
- Overriding is the basis for dynamic method dispatch – a call to an overridden method is resolved at run-time, rather than compile-time.
- Method overriding allows for dynamic method invocation:

1) an overridden method is called through the super-class variable
2) Java determines which version of that method to execute based on the type of the referred object at the time the call occurs
3) when different types of objects are referred, different versions of the overridden method will be called.

Example: Dynamic Invocation

- A super-class A:
- class A {

      void callme() {

  System.out.println("Inside A's callme method");

      }
- }

Example: Dynamic Invocation

- Two sub-classes B and C:
- class B extends A {

      void callme() {
      System.out.println("Inside B's callme method");
      }
- }
- class C extends A {

      void callme() {
      System.out.println("Inside C's callme method");
      }
- }
- B and C override the A’s callme() method.

Example: Dynamic Invocation

- Overridden method is invoked through the variable of the super-class type.
- Each time, the version of the callme() method executed depends on the type of the object being referred to at the time of the call:
- class Dispatch {
- public static void main(String args[]) {

  A a = new A();

  B b = new B();

  C c = new C();
- A r;
- r = a; r.callme();
- r = b; r.callme();
- r = c; r.callme();
- } }

Uses of final

- The final keyword has three uses:

1) declare a variable which value cannot change after initialization
2) declare a method which cannot be overridden in sub-classes
3) declare a class which cannot have any sub-classes

Uses of final

- Java final variable
- When a variable is declared with final keyword, its value can’t be modified, essentially, a constant. This also means that you must initialize a final variable.
- We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an initializer or an assignment statement.
- class FinalVariable {

      final int var = 50;

      var = 60 //This line would give an error
- }

Preventing Overriding with final

- A method declared final cannot be overridden in any sub-class:

```
class A {
        final void meth() {
                System.out.println("This is a final method.");
        }
}
```

- This class declaration is illegal:

```
class B extends A {
        void meth() {
                System.out.println("Illegal!");
        }
}
```

final and Early Binding

- Two types of method invocation:

1) early binding – method call is decided at compile-time
2) late binding – method call is decided at run-time

- By default, method calls are resolved at run-time.
- As a final method cannot be overridden, their invocations are resolved at compile-time.
- This is one way to improve performance of a method call.

Preventing Inheritance with final

- A class declared final cannot be inherited – has no sub-classes.
- final class A { … }
- This class declaration is considered illegal:
- class B extends A { … }
- Declaring a class final implicitly declares all its methods final.
- It is illegal to declare a class as both abstract and final.

Abstract Class

- A class that is declared using “abstract” keyword is known as abstract class.
- It can have abstract methods(methods without body) as well as concrete methods (regular methods with body).
- A normal class(non-abstract class) cannot have abstract methods.
- But, if a class has at least one abstract method, then the class must be declared abstract.
- If a class is declared abstract, it cannot be instantiated.

Abstract Class

- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstract Method

- Inheritance allows a sub-class to override the methods of its super-class.
- In fact, a super-class may altogether leave the implementation details of a method and declare such a method abstract:
- abstract type name(parameter-list);
- Two kinds of methods:
- 1) concrete – may be overridden by sub-classes
- 2) abstract – must be overridden by sub-classes
- It is illegal to define abstract constructors or static methods.

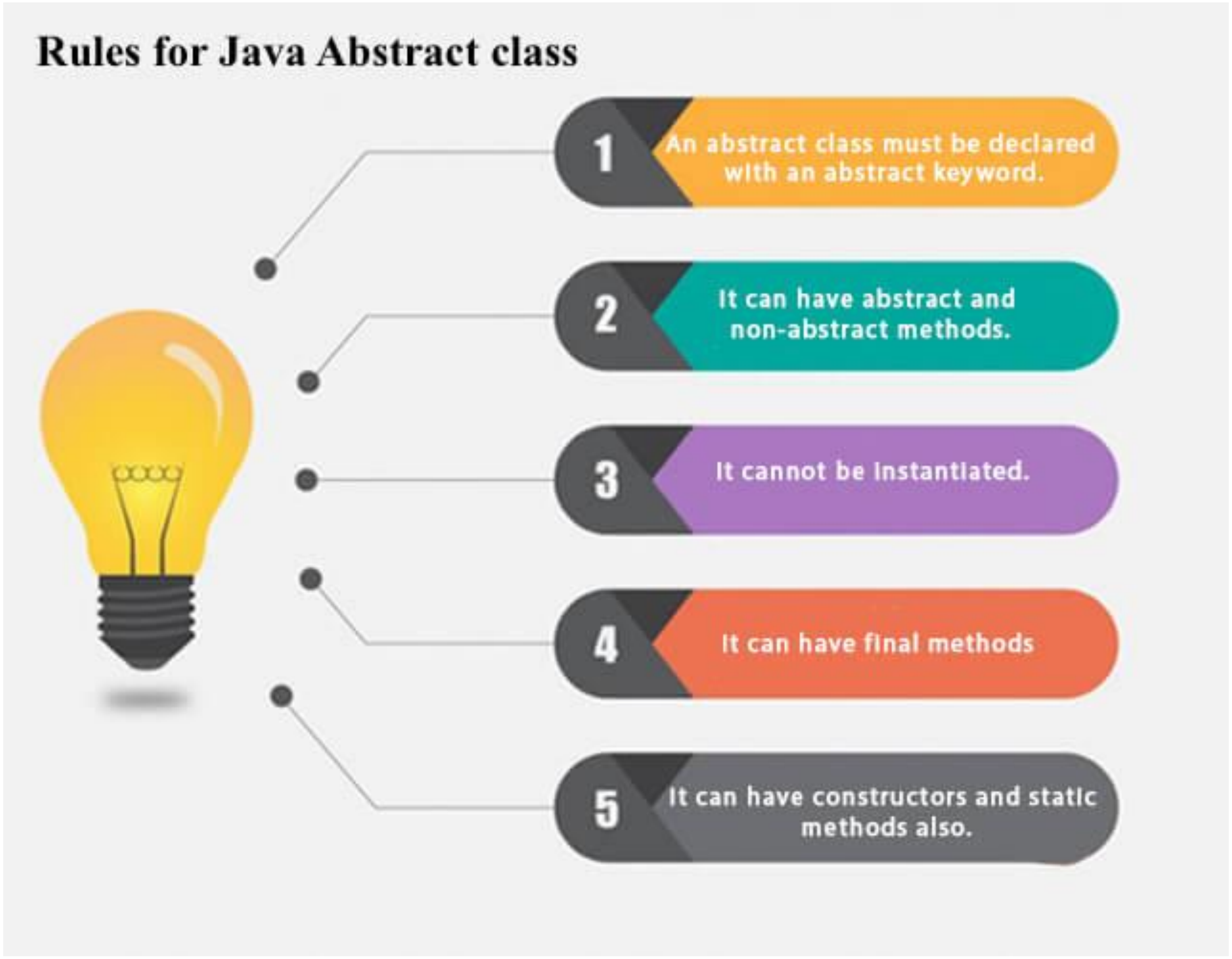
Abstract Class

- A class that contains an abstract method must be itself declared abstract:
- abstract class abstractClassName {

```
        abstract type methodName(parameter-list) {
        …
        }
…
```

- }
- An abstract class has no instances - it is illegal to use the new operator:
- ~~abstractClassName a = new abstractClassName();~~
- It is legal to define variables of the abstract class type.

Abstract Class



Rules for Java Abstract class

1. An abstract class must be declared with an abstract keyword.
2. It can have abstract and non-abstract methods.
3. It cannot be instantiated.
4. It can have final methods
5. It can have constructors and static methods also.

Abstract Sub-Class

- A sub-class of an abstract class:

1) implements all abstract methods of its super-class, or

2) is also declared as an abstract class

```
• abstract class A {
        abstract void callMe();
  }
• abstract class B extends A {
        int checkMe;
  }
```

Abstract and Concrete Classes

- Abstract super-class, concrete sub-class:

```
• abstract class A {
        abstract void callme();
        void callmetoo() {
                System.out.println("This is a concrete method.");
        }
• }
• class B extends A {
        void callme() {
                System.out.println("B's implementation.");
        }
• }
```

Abstract and Concrete Classes

- Calling concrete and overridden abstract methods:

```
• class AbstractDemo {
      public static void main(String args[]) {
              B b = new B();
              b.callme();
              b.callmetoo();
      }
• }
```

Example: Abstract Class

- Figure is an abstract class; it contains an abstract area method:
- abstract class Figure {

```
        double dim1;
        double dim2;
        Figure(double a, double b) {
                dim1 = a; dim2 = b;
        }
        abstract double area();
```

- }

Example: Abstract Class

- Rectangle is concrete – it provides a concrete implementation for area:

```
class Rectangle extends Figure {
	Rectangle(double a, double b) {
		super(a, b);
	}
	double area() {
	System.out.println("Inside Area for Rectangle.");
		return dim1 * dim2;
	}
```

- }

Example: Abstract Class

- Triangle is concrete – it provides a concrete implementation for area:

-
  ```
  class Triangle extends Figure {
          Triangle(double a, double b) {
                  super(a, b);
          }
          double area() {
                  System.out.println("Inside Area for Triangle.");
                  return dim1 * dim2 / 2;
          }
  ```
- }

Example: Abstract Class

- Invoked through the Figure variable and overridden in their respective subclasses, the area() method returns the area of the invoking object:
- class AbstractAreas {

  ```
      public static void main(String args[]) {
              Rectangle r = new Rectangle(9, 5);
              Triangle t = new Triangle(10, 8);
              Figure figref;
              figref = r; System.out.println(figref.area());
              figref = t; System.out.println(figref.area());
      }
  ```
- }

Abstract Class References

- It is illegal to create objects of the abstract class:
- Figure ~~f = new Figure(10, 10);~~
- It is legal to create a variable with the abstract class type:
- Figure figref;
- Later, figref may be used to assign references to any object of a concrete sub-class of Figure (e.g. Rectangle) and to invoke methods of this class:
- Rectangle r = new Rectangle(9, 5);
- figref = r; System.out.println(figref.area());

Example: Abstract Method

- The area method cannot compute the area of an arbitrary figure:

```
double area() {
	System.out.println("Area is undefined.");
	return 0;
}
```

Instead, area should be defined abstract in Figure:

```
abstract double area() ;
```

Example: Abstract Method

- Points to remember about abstract method:

1. Abstract method has no body.
2. Always end the declaration with a semicolon(;).
3. It must be overridden. An abstract class must be extended and in a same way abstract method must be overridden.
4. Abstract method must be in a abstract class.