Classes and interfaces

Yulia Newton, Ph.D.

CS151, Object Oriented Programming

San Jose State University

Spring 2020

Prior to object oriented programming

- Procedural programming
  - Functions operate on the data
  - Simpler to understand and implement … but
  - Much copying and pasting
  - No centralized code to make a single change to
  - Spaghetti code

Classes

- Class – basic unit of object oriented programming
  - Blueprint for an object
  - Abstraction of a given problem
  - A collection of attributes and methods and relationships with other objects
- Good class design provides robust class structure and functionality with longevity much longer than the immediate need/use
- A well designed class is correct to requirements, reusable, extensible, and easy to maintain
- Class instance – a particular realization of a class
  - Employee class
    - John Smith is an instance of Employee class
    - Jane Johnson is another instance of Employee class

Example of a class declaration

```
class Rectangle{
    private int length;
    private int width;

    public void rectArea(){
        int result;
        result = length * width;
        System.out.println("Area of rectangle is: " +result);
    }
}
```

Naming a class

- The name should reflect the purpose of the class
  - A noun
- Names should start with a capital letter (convention)
  - Can contain other capital letters anywhere in the name
- Names can be verbose
- Examples from Java API
  - *AbstractQueuedLongSynchronizer*, *AdapterNonExistentHelper*, *AlgorithmParameterGeneratorSpi, AnnotationTypeMismatchException, AtomicMoveNotSupportedException*

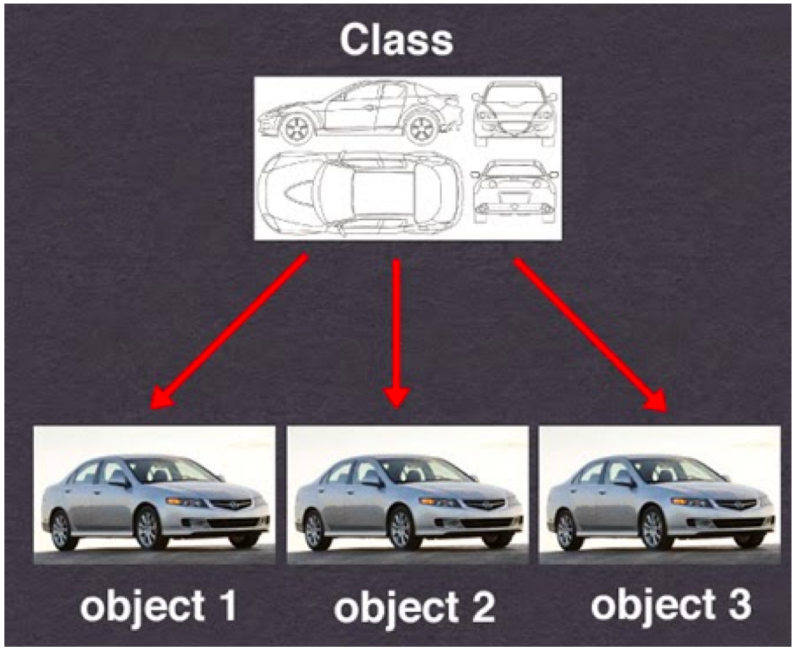
```
public class Employee{
    ...
}
```

Class components

- Attributes – variables/fields inside the class
  - Nouns
  - E.g. *name*, *id*, *address*
  - Set/updated and retrieved with setter and getter methods
- Methods – describe behaviors of the class
  - Verbs
  - E.g. *printInfo()*, *computeArea()*
- Constructors
  - Special methods used to initialize the objects
  - Called when an instance of the class is created
  - Class can have many constructors with different parameter lists

Class vs. object instance

- Class is a blueprint/prototype
- Instance is a unique entity of the class



https://www.java67.com/2014/11/difference-between-instance-and-object-in-java.html

Access modifiers

- Control visibility of a class, variable, or method
- All classes, attributes and methods can be declared
  - **Public** – can be accessed by any other class
  - **Protected** – can be accessed by subclasses in other packages or classes within the same package
  - **Private** – can only accessed by methods within this class
    - Classes and interfaces cannot be private
- If no access modifier is specified then “package protected” access (**default**)
  - Not visible or inherited outside of the package
  - Different from protected – not available to classes in other packages

Non-access modifiers

- **Static** – members declared as static are common to all instances of the class or interface
- **Final** – restricts further modifying of this variable, method, or class
  - Final variable cannot be modified once it gets a value
  - Final method cannot be overridden
  - Final class cannot be inherited
- **Abstract** – requires further modifications
  - Abstract method must be modified
  - Abstract class cannot be instantiated
  - Cannot be used with variables or constructor methods

Non-access modifiers (cont'd)

- **Synchronized** – only a single thread can enter into a method declared synchronized
- **Volatile** – variable common to all threads
- **Transient** – variable declared as transient has to be omitted during object serialization
- **Strictfp** – used for floating point calculations; makes floating point calculations platform independent
  - Results are consistent across platforms

Packages in java

- Way to encapsulate related classes
- Provide controlled access to classes and interfaces
- Prevent name conflicts
  - Two different packages may have classes with the same name (e.g. *university.cs.Employee* and *university.ee.Employee*
- Easier searching for and using classes and interfaces
- Naming is closely related to directory structure
  - *university.cs.Employee* – two levels of directories (university -> cs)
- Subpackages are packages inside other packages
  - Members of subpackages are considered as different package for protected and default access modifiers
  - *utils* is a subpackage inside *java* package:

```
import java.util.*;
```

Accessing classes inside java packages

- Using keyword *import*

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
```

Example:

```
import java.util.Scanner;

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);
    System.out.println("Enter username");

    String userName = myObj.nextLine();
    System.out.println("Username is: " + userName);
  }
}
```

https://www.w3schools.com

Types of Java packages

- User defined
  - Defined by the programmer
- Built in
  - Provided by Java API
  - https://docs.oracle.com/javase/8/docs/api/index.html?overview-summary.html

Useful built-in Java packages

- **java.lang** – language support, primitive data types, mathematical operations
  - Automatically imported
- **java.io** – input/output operations
- **java.utils** – utility classes (e.g. data structures)
- **java.awt** – classes for implementing graphical user interface
- **java.net** – classes that help with network operations

Creating a new package

- Create a directory with the name of your package
- Add classes to this directory
  - Use *package* keyword

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

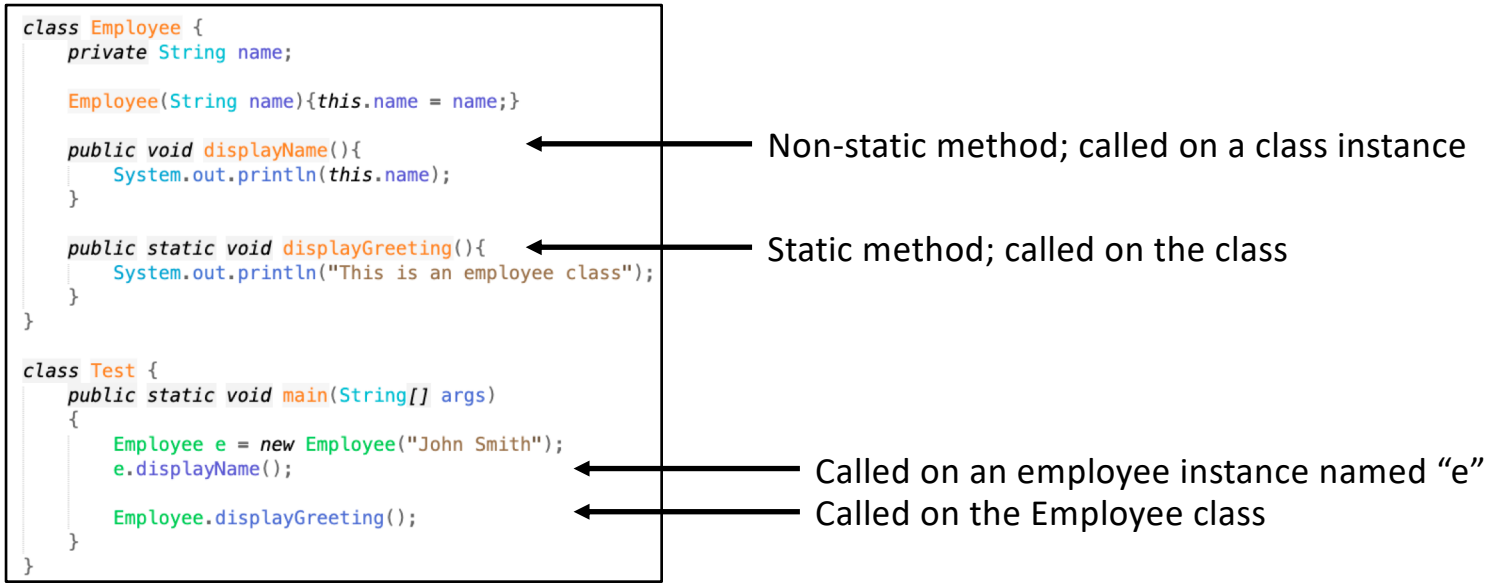
```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();

      obj.getNames(name);
   }
}
```

https://www.geeksforgeeks.org

Updating an existing package

- To add a class to a package just add a new *.java* file to the package directory
  - Use *package* keyword

Static vs. non-static class methods



Static vs. non-static class methods

Java compiler allows this:

```
class Employee {
    private String name;

    Employee(String name){this.name = name;}

    public void displayName(){
        System.out.println(this.name);
    }

    public static void displayGreeting(){
        System.out.println("This is an employee class");
    }
}

class Test {
    public static void main(String[] args)
    {
        Employee e = new Employee("John Smith");
        e.displayName();

        e.displayGreeting();
    }
}
```

Static vs. non-static class methods

Java compiler allows this:

```
class Employee {
    private String name;

    Employee(String name){this.name = name;}

    public void displayName(){
        System.out.println(this.name);
    }

    public static void displayGreeting(){
        System.out.println("This is an employee class");
    }
}

class Test {
    public static void main(String[] args)
    {
        Employee e = new Employee("John Smith");
        e.displayName();

        e.displayGreeting();
    }
}
```

Java compiler does NOT allow this:

```
class Employee {
    private String name;

    Employee(String name){this.name = name;}

    public void displayName(){
        System.out.println(this.name);
    }

    public static void displayGreeting(){
        System.out.println("This is an employee class");
    }
}

class Test {
    public static void main(String[] args)
    {
        Employee e = new Employee("John Smith");
        Employee.displayName();

        Employee.displayGreeting();
    }
}
```

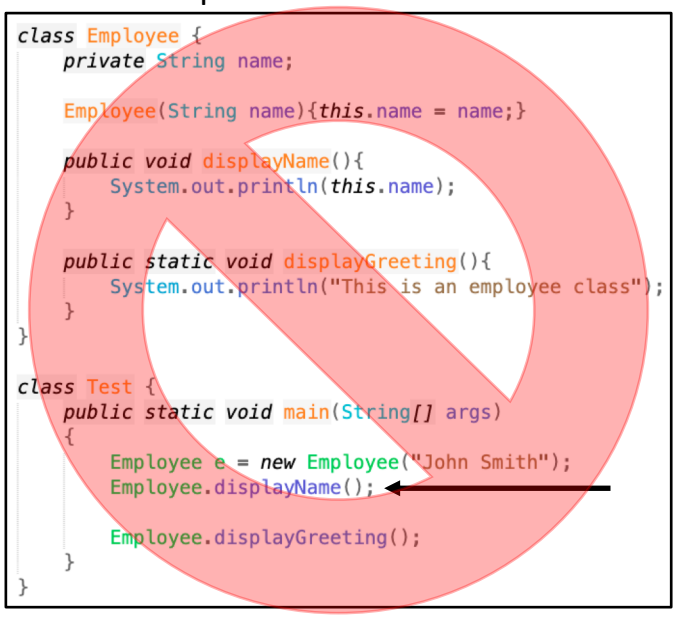
```
Test.java:12121: error: non-static method displayName() cannot be referenced from a static context
        Employee.displayName();
                 ^
1 error
```

Static vs. non-static class methods

Java compiler allows this:

```java
class Employee {
    private String name;

    Employee(String name){this.name = name;}

    public void displayName(){
        System.out.println(this.name);
    }

    public static void displayGreeting(){
        System.out.println("This is an employee class");
    }
}

class Test {
    public static void main(String[] args)
    {
        Employee e = new Employee("John Smith");
        e.displayName();

        e.displayGreeting();
    }
}
```

Java compiler does NOT allow this:



```java
class Employee {
    private String name;

    Employee(String name){this.name = name;}

    public void displayName(){
        System.out.println(this.name);
    }

    public static void displayGreeting(){
        System.out.println("This is an employee class");
    }
}

class Test {
    public static void main(String[] args)
    {
        Employee e = new Employee("John Smith");
        Employee.displayName();

        Employee.displayGreeting();
    }
}
```

```
Test.java:12121: error: non-static method displayName() cannot be referenced from a static context
        Employee.displayName();
                 ^
1 error
```

Example: static variables

```
class MathConstants {
    public static Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```

Example: static variables

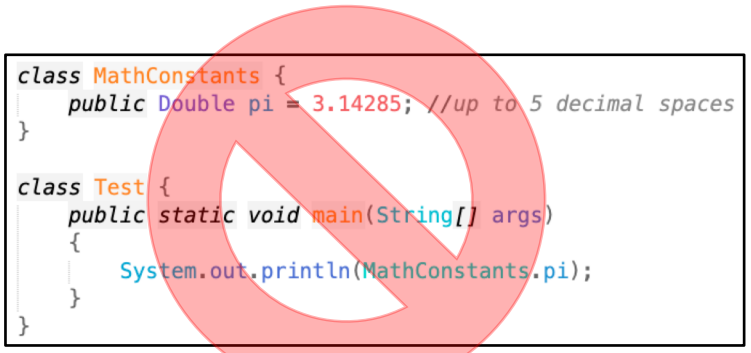
```java
class MathConstants {
    public static Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```

```java
class MathConstants {
    public Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```

```
Test.java:12110: error: non-static variable pi cannot be referenced from a static context
        System.out.println(MathConstants.pi);
                                         ^
1 error
```

Example: static variables

```java
class MathConstants {
    public static Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```

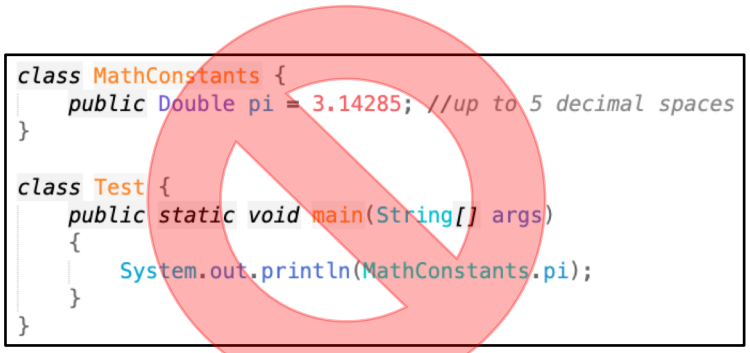


```java
class MathConstants {
    public Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```

```
Test.java:12110: error: non-static variable pi cannot be referenced from a static context
        System.out.println(MathConstants.pi);
                                         ^
1 error
```

Example: static variables

```java
class MathConstants {
    public static Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```



```java
class MathConstants {
    public Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        System.out.println(MathConstants.pi);
    }
}
```

```
Test.java:12110: error: non-static variable pi cannot be referenced from a static context
        System.out.println(MathConstants.pi);
                                         ^
1 error
```

```java
class MathConstants {
    public Double pi = 3.14285; //up to 5 decimal spaces
}

class Test {
    public static void main(String[] args)
    {
        MathConstants mc = new MathConstants();
        System.out.println(mc.pi);
    }
}
```

Static imports

- Using *static* keyword after *import* allows importing to allow unqualified access to static members of the package/class
- Analogous to normal import operations
- Use conservatively!

```
// Java Program to illustrate
// calling of predefined methods
// with static import
import static java.lang.Math.*;
class Test2 {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
// Java to illustrate calling of static member of
// System class without Class name
import static java.lang.Math.*;
import static java.lang.System.*;
class Geeks {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }
}
```

https://www.geeksforgeeks.org

Let’s examine Java API for these classes

- *java.lang* package
  - https://docs.oracle.com/javase/8/docs/api/index.html
- Math class in *java.lang* package
  - https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html
- System class in *java.lang* package
  - https://docs.oracle.com/javase/8/docs/api/java/lang/System.html

Example: *Math* class static import

```
public class Test {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }
}
```

Example: *Math* class static import

```java
public class Test {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }
}
```

```java
public class Test {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
Test.java:12104: error: cannot find symbol
        System.out.println(sqrt(4));
                           ^
  symbol:   method sqrt(int)
  location: class Test
Test.java:12105: error: cannot find symbol
        System.out.println(pow(2, 2));
                           ^
  symbol:   method pow(int,int)
  location: class Test
Test.java:12106: error: cannot find symbol
        System.out.println(abs(6.3));
                           ^
  symbol:   method abs(double)
  location: class Test
3 errors
```

Example: *Math* class static import

```java
public class Test {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }
}
```

```java
public class Test {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```java
import java.lang.Math.*;

public class Test {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
Test.java:12104: error: cannot find symbol
        System.out.println(sqrt(4));
                           ^
  symbol:   method sqrt(int)
  location: class Test
Test.java:12105: error: cannot find symbol
        System.out.println(pow(2, 2));
                           ^
  symbol:   method pow(int,int)
  location: class Test
Test.java:12106: error: cannot find symbol
        System.out.println(abs(6.3));
                           ^
  symbol:   method abs(double)
  location: class Test
3 errors
```

Example: *Math* class static import

```java
public class Test {
    public static void main(String[] args)
    {
        System.out.println(Math.sqrt(4));
        System.out.println(Math.pow(2, 2));
        System.out.println(Math.abs(6.3));
    }
}
```

```java
public class Test {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
Test.java:12104: error: cannot find symbol
        System.out.println(sqrt(4));
                           ^
  symbol:   method sqrt(int)
  location: class Test
Test.java:12105: error: cannot find symbol
        System.out.println(pow(2, 2));
                           ^
  symbol:   method pow(int,int)
  location: class Test
Test.java:12106: error: cannot find symbol
        System.out.println(abs(6.3));
                           ^
  symbol:   method abs(double)
  location: class Test
3 errors
```

```java
import java.lang.Math.*;

public class Test {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```java
import static java.lang.Math.*;

public class Test {
    public static void main(String[] args)
    {
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
2.0
4.0
6.3
```

Example: *System* class static import

```java
import static java.lang.Math.*;

class Test {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
2.0
4.0
6.3
```

Example: *System* class static import

```java
import static java.lang.Math.*;

class Test {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
2.0
4.0
6.3
```

```java
import static java.lang.Math.*;

class Test {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }
}
```

```
Test.java:12108: error: cannot find symbol
        out.println(sqrt(4));
        ^
  symbol:   variable out
  location: class Geeks
Test.java:12109: error: cannot find symbol
        out.println(pow(2, 2));
        ^
  symbol:   variable out
  location: class Geeks
Test.java:12110: error: cannot find symbol
        out.println(abs(6.3));
        ^
  symbol:   variable out
  location: class Geeks
3 errors
```

Example: *System* class static import

```java
import static java.lang.Math.*;

class Test {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        System.out.println(sqrt(4));
        System.out.println(pow(2, 2));
        System.out.println(abs(6.3));
    }
}
```

```
2.0
4.0
6.3
```

```java
import static java.lang.Math.*;
import static java.lang.System.*;

class Test {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }
}
```

```java
import static java.lang.Math.*;

class Test {
    public static void main(String[] args)
    {
        // We are calling static member of System class
        // directly without System class name
        out.println(sqrt(4));
        out.println(pow(2, 2));
        out.println(abs(6.3));
    }
}
```

```
Test.java:12108: error: cannot find symbol
        out.println(sqrt(4));
        ^
  symbol:   variable out
  location: class Geeks
Test.java:12109: error: cannot find symbol
        out.println(pow(2, 2));
        ^
  symbol:   variable out
  location: class Geeks
Test.java:12110: error: cannot find symbol
        out.println(abs(6.3));
        ^
  symbol:   variable out
  location: class Geeks
3 errors
```

Ambiguity in static imports

- If multiple static members with the same name are imported from multiple classes then compile-time error occurs

```
import static java.lang.Integer.*;
import static java.lang.Long.*;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println(MAX_VALUE);
    }
}
```

MAX_VALUE is an attribute in both *java.lang.Integer* and *java.lang.Long* classes

https://en.wikipedia.org/wiki/Static_import

Primitive vs. non-primitive class members

- Each class member/attribute has a data type
  - Primitive data types (e.g. boolean, int, char, byte, short, long, float, double)
    - Considered simple data types that hold a single value
  - Non-primitive data types (e.g. String, Array, user-defined classes, etc.)

- There is a trade off
  - Primitive data types are faster to work with but do not have special capabilities
  - Non-primitive data types can take up more resources

int vs. Integer types in Java

- *int* is a primitive data type
- *Integer* is a class in *Java.lang.Number* package
  - Wrapper class for integer values
  - Includes methods and functionality that do not come with *int* variables

```java
public class Test {
    public static void main(String args[]){
        int primitiveInt = 200;
        System.out.println("primitiveInt = "+primitiveInt);

        Integer nonprimitiveInt = new Integer("300");
        System.out.println("nonprimitiveInt = "+nonprimitiveInt);

        String binary = Integer.toBinaryString(400);
        System.out.println("400 as binary = "+binary);
        String oct = Integer.toOctalString(400);
        System.out.println("400 as oct = "+oct);
        String hex = Integer.toHexString(400);
        System.out.println("400 as hex = "+hex);

        // static parseInt() method can assist in declaring primitive int
        int anotherPrimitiveInt = Integer.parseInt("500");
        System.out.println("anotherPrimitiveInt = "+anotherPrimitiveInt);
    }
}
```

Output:

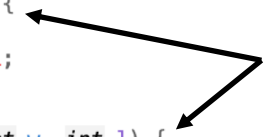
```
primitiveInt = 200
nonprimitiveInt = 300
400 as binary = 110010000
400 as oct = 620
400 as hex = 190
anotherPrimitiveInt = 500
```

Class constructors

- Special method that gets called when an object of the class is created
  - Often used to set initial data values
- If no constructor is provided by the class designer/programmer Java compiler automatically provides default constructor
  - All values are initialized to whatever default values are for the variable types
- Multiple constructors can be provided
  - Different arguments and argument types

Example: class constructors

```
class Rectangle {
    private int width, length;

    public Rectangle() {
        this.width = 1;
        this.length = 1;
    }

    public Rectangle(int w, int l) {
        this.width = w;
        this.length = l;
    }

    public void display() {
        System.out.println("Width = " + this.width + ", length = " + this.length);
    }
}

public class Test
{
    public static void main(String args[])
    {
        Rectangle rect1 = new Rectangle();
        rect1.display();

        Rectangle rect2 = new Rectangle(5,10);
        rect2.display();
    }
}
```




```
Width = 1, length = 1
Width = 5, length = 10
```

Instance initializer block

- Used to initialize instance data members
- Run each time an instance of the class is created
- Might want to use instance initializer block because want to perform other operations while assigning values to data fields
  - In Java operations can be performed in:
    - Methods
    - Constructors
    - Initializer blocks
- Syntactically, wrapped into *{…}*

Example: instance initializer block

```
class Animal
{
    int numLimbs;
    Animal(){System.out.println("Animal constructor is invoked");}

    {System.out.println("Animal instance initializer block invoked");}
}

public class Test
{
    public static void main(String args[])
    {
        Animal a1 = new Animal();
        Animal a2 = new Animal();
    }
}
```

instance initializer block is invoked before the constructor body

Output:

```
Animal instance initializer block invoked
Animal constructor is invoked
Animal instance initializer block invoked
Animal constructor is invoked
```

instance initializer block

How initializer blocks are invoked

- Java compiler copies instance initializer block code into every constructor
- Instance initializer block code is executed whenever an instance of the class is created
- Invoked after the the parent class constructor code is invoked
  - *super()* constructor call

Example: instance initializer block in both child and parent classes

```
class Animal
{
    Animal(){System.out.println("Animal constructor is invoked");}

    {System.out.println("Animal instance initializer block invoked");}
}

class Cat extends Animal
{
    Cat(){
        super();
        System.out.println("Cat constructor is invoked");
    }

    {System.out.println("Cat instance initializer block invoked");}
}

public class Test
{
    public static void main(String args[])
    {
        Cat c1 = new Cat();
    }
}
```

Output:

```
Animal instance initializer block invoked
Animal constructor is invoked
Cat instance initializer block invoked
Cat constructor is invoked
```

Class setters and getters

- Setter methods – set and update values of attributes via setters
- Getter methods – retrieve values of attributes via getters
- Good object oriented programming practice!

Example: setters and getters in the Employee class

Without using setters/getters:

```
class Employee{
   public String name;
   public String phone;
   public int id;
}

public class test {
    public static void main(String args[])
    {
        Employee e1 = new Employee();
        e1.name = "Steve";
        e1.phone = "123-456-7890";
        e1.id = 101;

        System.out.println(e1.id + "; " + e1.name + "(" + e1.phone + ")");
    }
}
```

Output: `101; Steve(123-456-7890)`

Using setters/getters:

```
class Employee{
   private String name;
   private String phone;
   private int id;

   public String getName() { return name; }
   public void setName(String name)
      { this.name=name; }

   public String getPhone() { return phone; }
   public void setPhone(String phone)
      { this.phone=phone; }

   public int getID() { return id; }
   public void setID(int id)
      { this.id=id; }
}

public class test {
    public static void main(String args[])
    {
        Employee e1 = new Employee();
        e1.setName("Steve");
        e1.setPhone("123-456-7890");
        e1.setID(101);

        System.out.println(e1.getID() + "; " + e1.getName() + "(" + e1.getPhone() + ")");
    }
}
```

Good ideas for class design

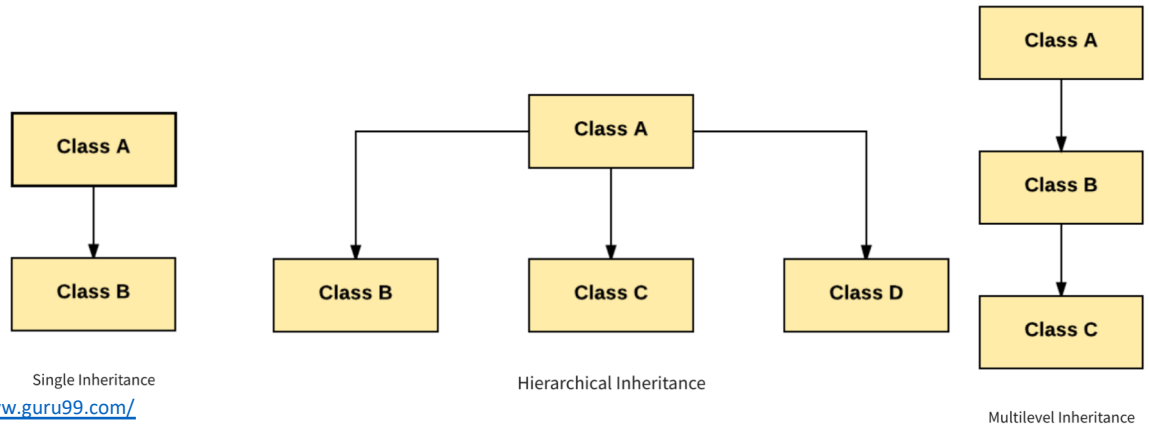
- Data should be made private
  - Use public methods to access and manipulate data
- Internal utility methods should be private
- Provide setters and getters
  - Setters and getters should be separate methods
- Always provide at least one constructor
  - Do not rely on the default constructor
- All constructors should initialize the object to a valid state
- All mutator methods should leave the object in a valid state
- Cannot have multiple public class declarations in the same file
  - Create separate files for each class in your program

Composition is a fundamental concept in OOD

- A single class should follow ”single responsibility principle”
- When designing the system ask “is the child a type of parent?”
  - If the answer is yes, use inheritance
  - If the answer is no or somewhat, use composition
- Composition allows using some of the functionality of the class without inheriting from it
  - “Has-a” relationship

Inheritance

- A class can inherit from another class
- “Is-a” relationship
- Inheritance indicated with a keyword *extends*
- Child class inherits all public methods and functionality of a parent class



https://www.guru99.com/

```
public class Animal {
    ...
}

public class Dog extends Animal {
    ...
}

public class Cat extends Animal {
    ...
}

public class DomesticatedCat extends Cat {
    ...
}

public class FeralCat extends Cat {
    ...
}
```

Super keyword

- Refers to the immediate parent class object
- When an instance of a child class is created and instance of the parent class is implicitly created as well
  - This instance of the parent class is referred to as *super*

Example: using *super* with variables

```java
class Pet
{
    public int numLegs = 4;
}

class Tarantula extends Pet {
    private int numLegs = 8;

    void display()
    {
        System.out.println("Number of tarantula legs: " + this.numLegs);
        System.out.println("Number of generic pet legs: " + super.numLegs);
    }
}

public class Test {
    public static void main(String args[])
    {
        Tarantula tarantula = new Tarantula();
        tarantula.display();
    }
}
```

```
Number of tarantula legs: 8
Number of generic pet legs: 4
```

Example: using *super* with methods

```java
class Animal
{
    public void introduce()
    {
        System.out.println("I am an animal");
    }
}

class Cat extends Animal
{
    public void introduce()
    {
        System.out.println("I am a cat");
    }

    public void display()
    {
        this.introduce();
        super.introduce();
    }
}

public class Test
{
    public static void main(String args[])
    {
        Cat cat = new Cat();
        cat.display();
    }
}
```

```
I am a cat
I am an animal
```