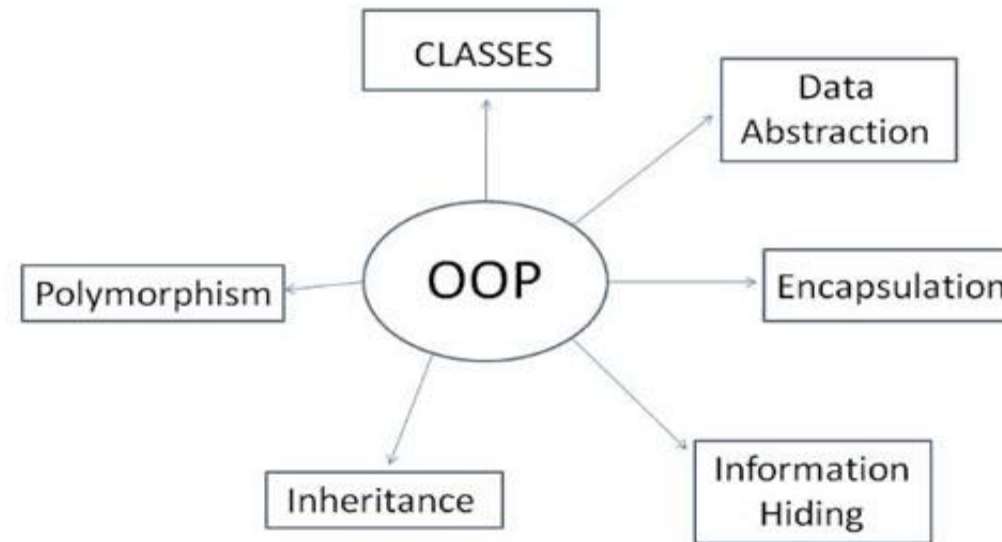


Object Oriented Programming Concept (OOP)





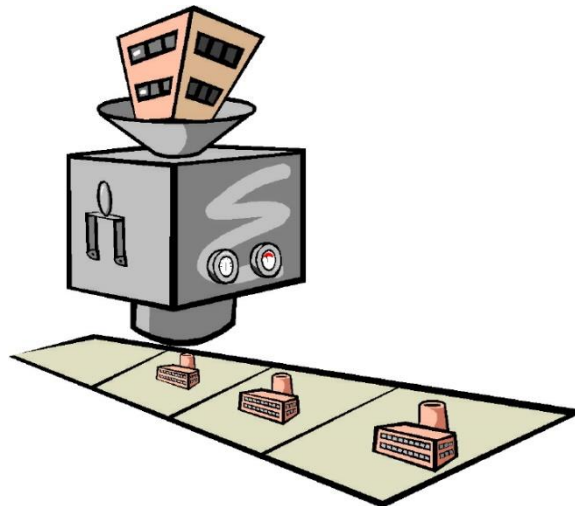
Outlines

- › Object Oriented Programming Concept (OOP)
- › Class
- › Object Instantiation
- › Using object data/methods
- › Method
- › Constructor
- › Keywords “this”, “static”, “final”
- › Package
- › Method in more details
 - Access modifiers, Getter & setters, Passing method arguments
 - toString(), equals()



Object Oriented Programming Concept (OOP)

- › An *object* is a data that can contain other data inside.
- › An *object-oriented program* works with many objects.
- › A *class* is a definition (or blueprint) used to create objects of the same kind.
 - So, an *object* is an instance of a class.
- › We ask a class to create an object by using the keyword: *new* ClassName.





A new data of type Car.

```
class Car{  
    int speed;  
    int acc;  
}
```

Used in another
program (or another
class)

```
class TransportSim{  
    main(){  
        Car a = new Car();  
        Car b = new Car();  
        ...  
    }  
}
```



OOP concepts

- › View everything as objects and their interactions
- › Encapsulation/Information Hiding (Structure)
- › Abstraction (Structure)
- › Inheritance (Structure)
- › Polymorphism (Behavior)



Encapsulation/Information Hiding(Structure)

- › Hides all internal implementations from others. (safe from outside manipulation)
- › A class can change its internal implementation without effecting outside agents.
 - Outside agents can be forced to use only method(s) of the class
- › Accessibility: public, protected, **private**
- › How can we ensure that changing will not hurt? (Unit test)

Car
<ul style="list-style-type: none">- brand: String- model: String- engine: Engine- wheel: integer+ options:...
<ul style="list-style-type: none">+ start(): void+ move(direction): void...



Not allowed!

```
class Car{  
    private int speed;  
    private int acc;  
  
    public void pushAcc (){  
        speed = speed+acc;  
    }  
}
```

Allowed!

```
class TransportSim{  
    main(){  
        Car a = new Car();  
        Car b = new Car();  
        a.speed = -555;  
        a.pushAcc();  
    }  
}
```



Abstraction (Structure)

- › We can merge common codes!
 - Combine common characteristics to build class hierarchy
- › Reduce duplication



Abstraction

Car
<ul style="list-style-type: none"> - brand: String - model: String - engine: Engine - wheel: integer
<ul style="list-style-type: none"> + options: ...
<ul style="list-style-type: none"> + start(): void + move(direction): void ...

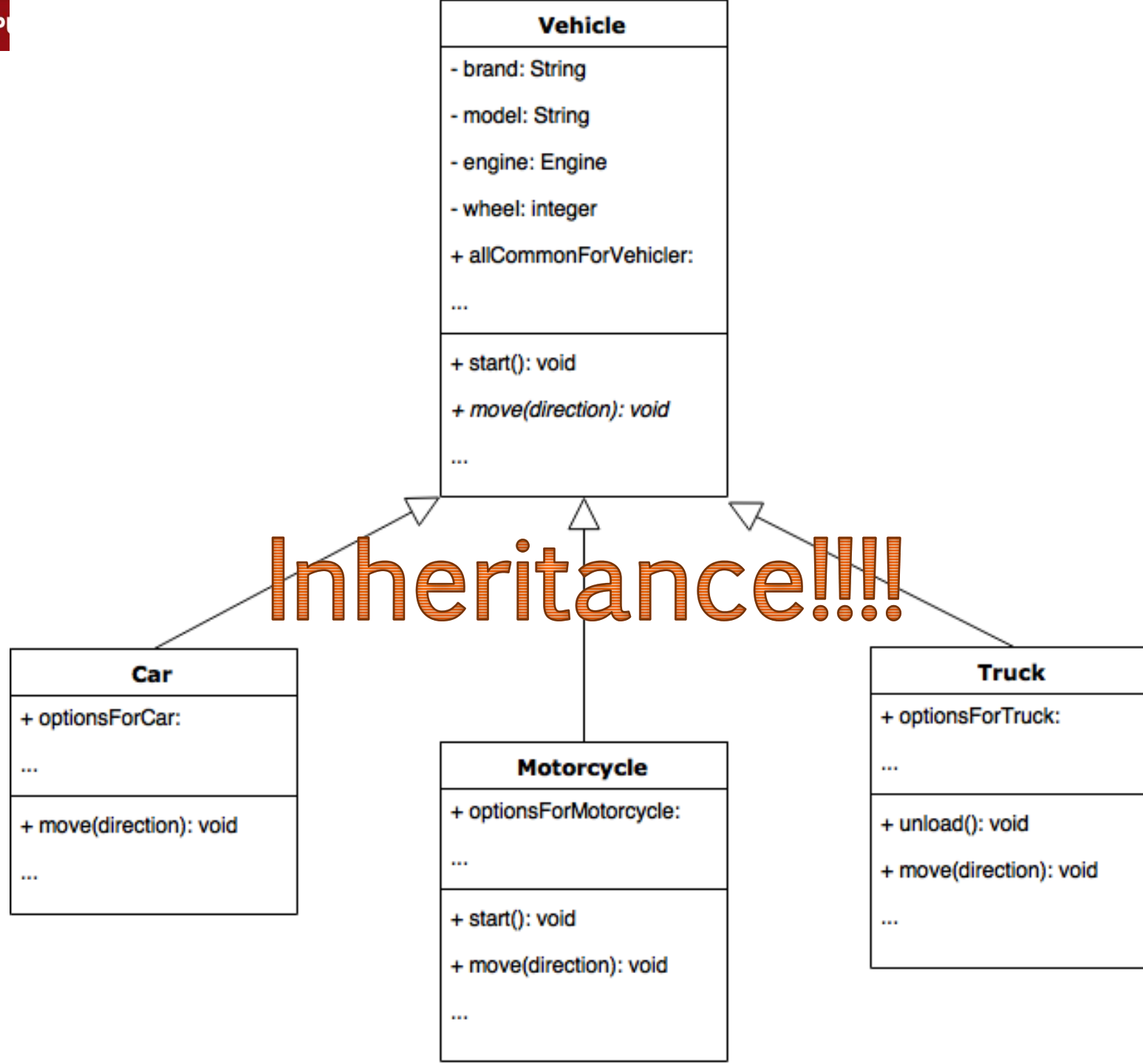


Truck
<ul style="list-style-type: none"> - brand: String - model: String - engine: Engine - wheel: integer
<ul style="list-style-type: none"> + optionsForTruck: ...
<ul style="list-style-type: none"> + start(): void + move(direction): void ...



Motorcycle
<ul style="list-style-type: none"> - brand: String - model: String - engine: Engine - wheel: integer
<ul style="list-style-type: none"> + optionsForMotorcycle: ...
<ul style="list-style-type: none"> + start(): void + move(direction): void ...

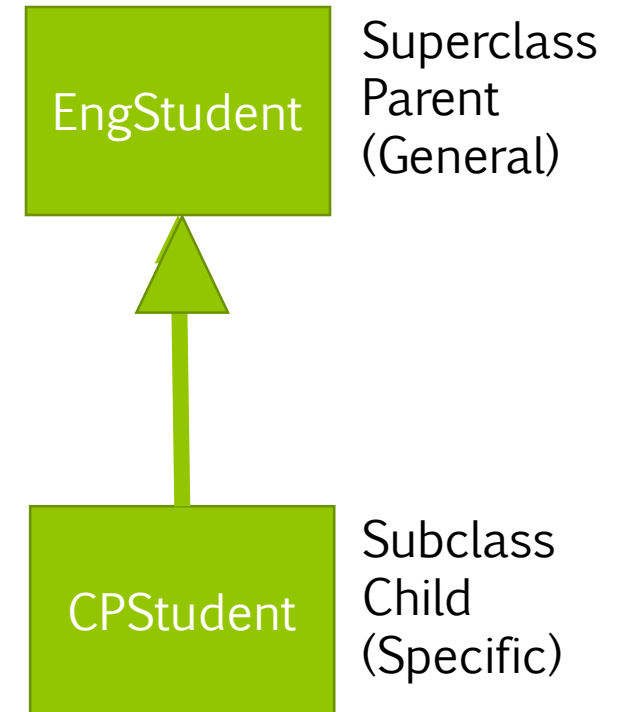






Inheritance (Structure)

- › Define subclasses from existing class
- › The subclass object “is-a” or ”is-a-type-of” superclass object
 - All Cars are also Vehicles. But a Vehicle does not have to be a Car.
 - All CP students are engineer student, but an engineering may be or may not be a CP student.
- › Vehicle is reusable as a superclass of future class(es).





Polymorphism (Behavior)

- › Different types of objects can receive the same command (method).
- › And each of them behave differently according to their actual type.
- › We can write a program that correctly works on all of those objects, using the same code.



Image from <http://www.c-sharpcorner.com/UploadFile/433c33/polymorphism-in-java/>



Classes

- › Description of objects that share same attributes/properties/fields (called data members) and actions/behaviors/methods (called member functions)
- › Template for creating/instantiating
- › Example: Car, Dice



Objects

- › instances of classes
- › An object has identity, state and behaviors
- › Examples:
 - Real world object: Car object, graphic objects (circle, square, ...)
 - Abstract entities: an opened file, a network connection, an object that provides the services for currency conversion



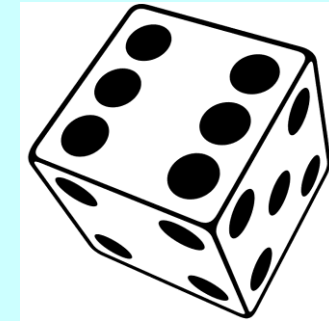
Class

- › A class contains 2 main components:
 - data (attributes, properties)
 - methods
 - › Constructor is a special method to create class object.






Class “SimpleDice”

```
public class SimpleDice {  
    final static int MAX = 6;  
    int faceValue;  
    public SimpleDice(int faceValue) {  
        this.faceValue = faceValue;  
    }  
    public int roll() {  
        faceValue = (int) (Math.random() * MAX) + 1;  
        return faceValue;  
    }  
}
```

Class “SimpleDice”

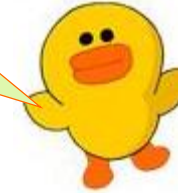


<<Java Class>>

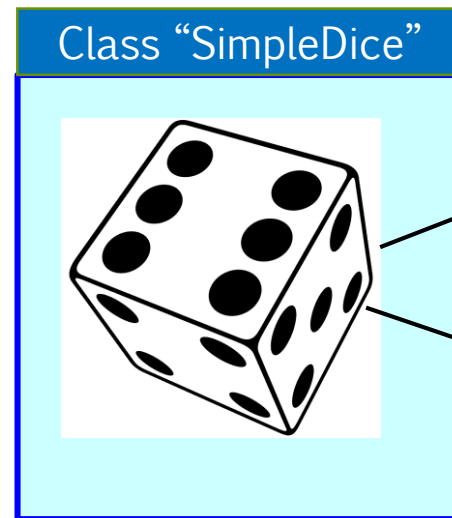
 **SimpleDice**
(default package) MAX: int faceValue: int SimpleDice(int) roll():int

Class UML diagram

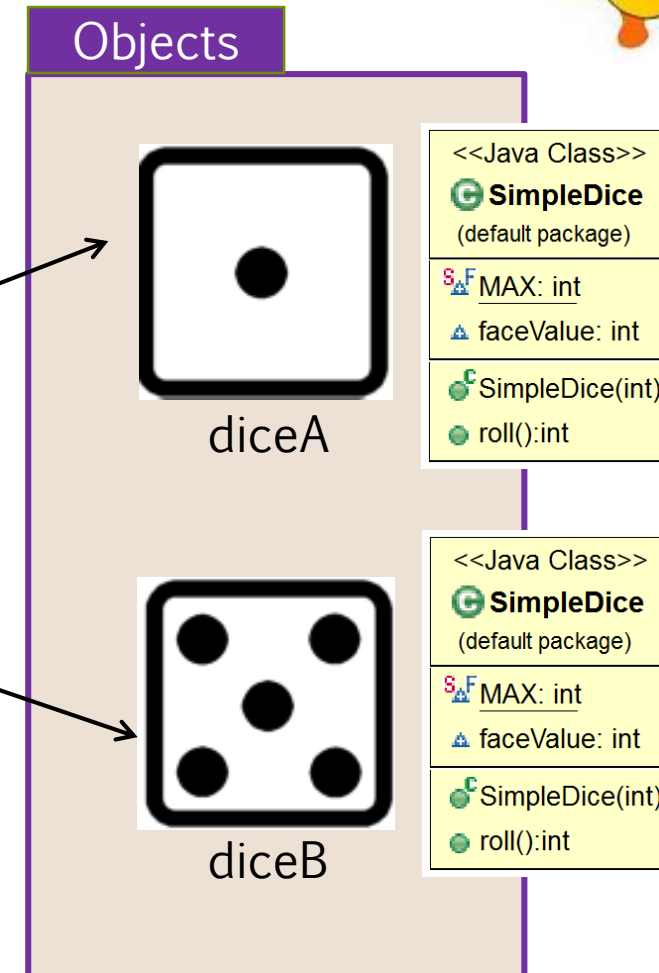
Important! Each object has its own data and methods.



Object Instantiation



instantiate



Code

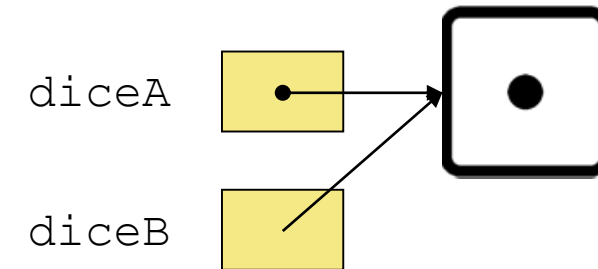
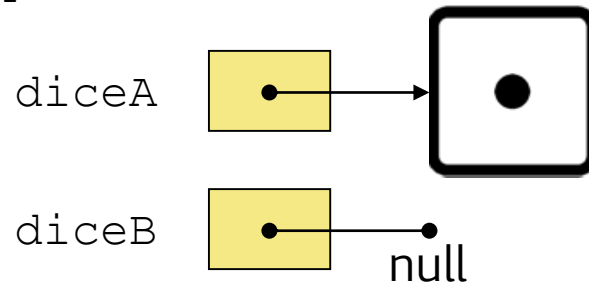
```
SimpleDice diceA = new SimpleDice(1);
SimpleDice diceB = new SimpleDice(5);
```



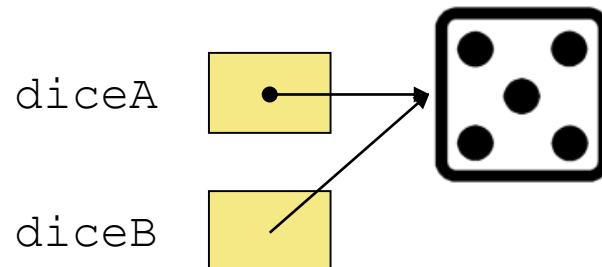

Object Instantiation (cont.)

1 `SimpleDice diceA = new SimpleDice(1);` 2 `diceB = diceA;`

`SimpleDice diceB;`



3 `diceB.faceValue = 5;`



Code

```
SimpleDice diceA = new SimpleDice(1);
SimpleDice diceB;
diceB = diceA;
diceB.faceValue = 5;
System.out.println(diceA.faceValue);
```

What is the result of this program?

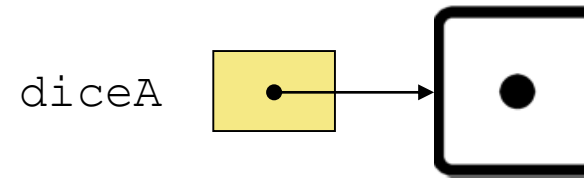




Using object data/methods

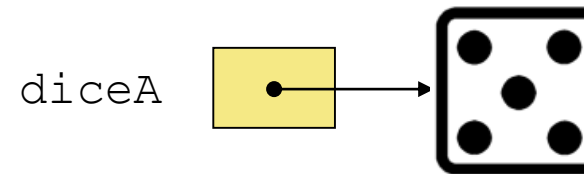
1. Create new object

```
SimpleDice diceA = new SimpleDice(1);
```



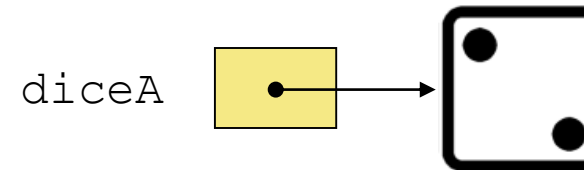
2. Access an object data

```
diceA.faceValue = 5;
```



3. Access an object method

```
System.out.println( diceA.roll() );
```



4. Access an object data (again)

```
System.out.println( diceA.faceValue );
```

What is the result of
the code in Step 4?










Method

- › A *method* is a small, well-defined piece of code that completes a specific task.
- › Typically, methods are used to change *variables (data)*.
 - For example, the method “roll” changes the value of “faceValue”.

Class “SimpleDice”

```
public int roll() {  
    faceValue = (int) (Math.random() * MAX) + 1;  
    return faceValue;  
}
```

<<Java Class>>

 **SimpleDice**
(default package) **MAX: int** **faceValue: int** **SimpleDice(int)** **roll():int**



Method Overloading

- › A class can have more than one method with the same name but must have unique signature

- › Method signature – name + arguments list
 - `add(int m, int n) → add(int, int)`
 - `add(double x, double y) → add(double, double)`
 - `add(int x, int y, int z) → add(int, int, int)`

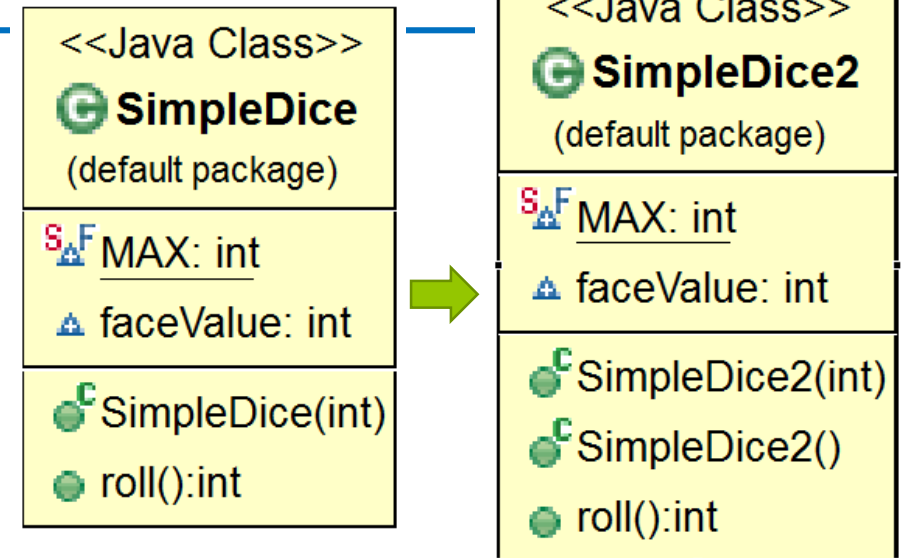


Constructor

- › It is a special type of method to define how to create an object.
- › It is called once when the object is created (before any other method will be invoked)
- › Constructors must:
 - Same name as class
 - Initializes instance variables of a class object
 - Called when program instantiates an object of that class
 - Can take arguments, but *cannot return values*
 - Class can have several constructors, through *overloading*

Class "SimpleDice"

```
public SimpleDice(int faceValue) {  
    this.faceValue = faceValue;  
}  
  
public SimpleDice() {  
    this.faceValue = this.roll();  
}
```





Keyword “this”

- › It allows an object to refers to itself.
- › It can be used in both class data and methods.
 - › this.faceValue, this.roll()
- › In Code1, it is necessary to use keyword “this” since the variable names are duplicated.
- › In Code2, it is **not** necessary to use keyword “this”.

Code1







```
public SimpleDice(int faceValue) {  
    this.faceValue = faceValue;  
}
```

Code2

```
public SimpleDice(int val) {  
    // faceValue = val  
    this.faceValue = val;  
}
```

Class “SimpleDice”

```
public SimpleDice(int faceValue) {  
    this.faceValue = faceValue;  
}  
  
public SimpleDice() {  
    this.faceValue = this.roll();  
}
```

<<Java Class>>	
	SimpleDice2 (default package)
	MAX: int
	faceValue: int
	SimpleDice2(int)
	SimpleDice2()
	roll():int









Keyword “this”

- › “this” as a method is a way to let a constructor calling other constructor.

Class “SimpleDice”

```
public SimpleDice(int faceValue) {  
    this.faceValue = faceValue;  
}  
  
public SimpleDice() {  
    this.faceValue = this.roll();  
}
```

<<Java Class>>	
 SimpleDice2 (default package)	
	MAX: int
	faceValue: int
 SimpleDice2(int)	
 SimpleDice2()	
 roll():int	

Class “SimpleDice”

```
public SimpleDice(int faceValue) {  
    this.faceValue = faceValue;  
}  
  
public SimpleDice() {  
    this(this.roll());  
}
```



Keyword “static”







- › The static keyword is used when a member variable of a class has to be shared between all the instances of the class.
- › It can be used at any levels: data/methods, class, and blocks
- › All static variables and methods **belong to the class** and not to any instance of the class
 - So, they can be directly invoked from “class”.

Example1: class “SimpleDice”

```
SimpleDice diceA = new SimpleDice(1);  
System.out.println( diceA.MAX );  
System.out.println( SimpleDice.MAX );
```

Example2: class “Math”

```
System.out.println( Math.PI );  
System.out.println( Math.random() );  
System.out.println( Math.floor(1.2) );
```

<<Java Class>>	
	SimpleDice2 (default package)
	MAX: int
	faceValue: int
	SimpleDice2(int)
	SimpleDice2()
	roll():int

Class “SimpleDice”

```
public class SimpleDice {  
    final static int MAX = 6;  
    int faceValue;  
}
```




Keyword “final”

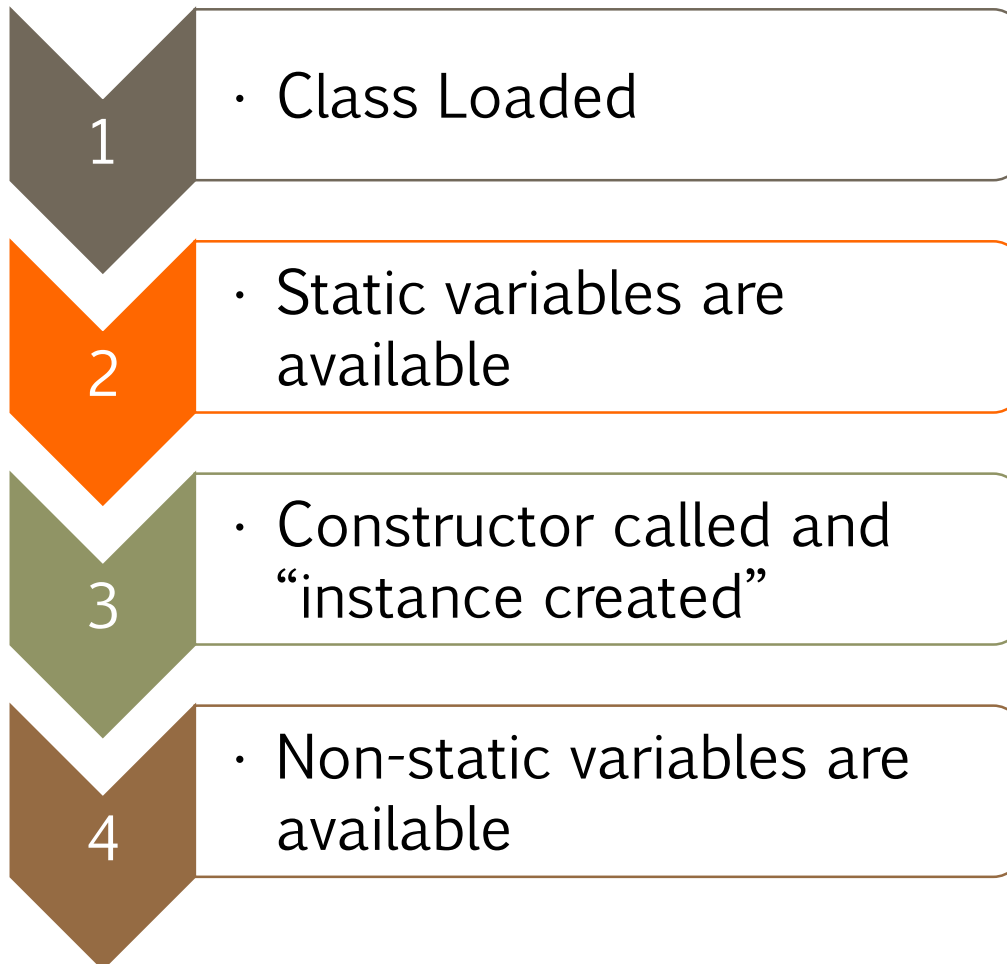
- › Specifies that a variable is **not** modifiable (is a constant)
- › It can be used at any levels: data/methods, class, and blocks
 - Final data: It means data cannot be modified or changed its value.
 - Final method: It means method cannot be overridden by subclasses.
 - Final class: It means class cannot be inherited.
- › For a constant value, it is commonly applied “final static”
 - For example, the variable “MAX” in the class “SimpleDice”

Class “SimpleDice”

```
public class SimpleDice {  
    final static int MAX = 6;  
    int faceValue;  
}
```



How to load class objects



- › Step1: Class is loaded by JVM
- › Step2: Static variable and methods are loaded and initialized and available for use
- › Step3: Constructor is called to instantiate the non static variables
- › Step4: Non-static variables and methods are now available



Package

- › A *package* is a collection of related classes and interfaces providing access protection and namespace management.
- › Java classes and interfaces are members of various packages that bundle classes by function:
 - fundamental classes are in **java.lang**,
 - classes for reading and writing (input and output) are in **java.io**,
 - etc.
- › Avoid namespace conflict

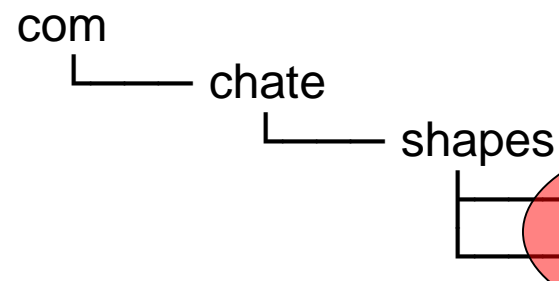


Package (cont.)

- › Use a **package** statement at the top of the source file in which the class or the interface is defined.

```
package com.chate.shapes;  
  
public class Oval {  
    // . . .  
}
```

```
package com.chate.shapes;  
  
public class Rectangle {  
    // . . .  
}
```



classes in the same
package are in the
same directory/folder



Package (cont.)

1. Full qualified name

```
com.chate.shapes.Oval o = new Oval();  
com.chate.shapes.Rectangle r = new Rectangle();
```

2. With import

```
import com.chate.shapes.Oval;  
  
import com.chate.shapes.Rectangle;  
  
// to import all members  
// import com.chate.shapes.*;  
  
Oval o = new Oval();  
  
Rectangle r = new Rectangle();
```



Method in more details

- › Access modifiers
- › Getter & setters
- › toString & equals
- › Passing method arguments



Access modifiers

Specifier	Class	Package	Subclass	World	UML Symbol
private	✓				-
package (default)	✓	✓			~
protected	✓	✓	✓		#
public	✓	✓	✓	✓	+









Class “Dice” with Encapsulation Concept

Class “Dice”

```
public class SimpleDice {  
    public final static int MAX = 6;  
    private int faceValue;  
    public SimpleDice(int faceValue) {  
        this.faceValue = faceValue;  
    }  
    public SimpleDice(int faceValue) {  
        this.faceValue = faceValue;  
    }  
    public int roll() {  
        faceValue = (int) (Math.random() * MAX) + 1;  
        return faceValue;  
    }  
}
```

<<Java Class>>

 **SimpleDice3**
(default package) MAX: int faceValue: int SimpleDice3(int) SimpleDice3() roll():int

- How can we access class data?
- Getter & Setter





Getters & Setters

Class "Dice"









```
public class SimpleDice {  
    public final static int MAX = 6;  
    private int faceValue;  
    public void setFaceValue(int value) {  
        faceValue = value;  
    }  
    public int getFaceValue() {  
        return faceValue;  
    }  
    ...  
}
```

> Getters

- public method
- Allow clients to read private data

> Setters

- public method
- Allow clients to modify private data

<<Java Class>>	
	SimpleDice3
(default package)	
	<u>MAX: int</u>
	faceValue: int
	SimpleDice3(int)
	SimpleDice3()
	roll():int
	getFaceValue():int
	setFaceValue(int):void



Basic methods

- › toString()
 - Convert object to a String representation

- › Identity test
 - Test whether two objects/references are the same actual object.
(default test when use “==”)

- › Equality test
 - Test whether two objects are logically equal.




Method toString() & equals()










Class "Dice"

```
public class SimpleDice {  
    ...  
    @Override  
    public String toString() {  
        return "SimpleDice4 [faceValue=" + faceValue + "]";  
    }  
  
    public boolean equals(Object o) {  
        Dice otherDice = (Dice) o;  
        if (this.getFaceValue() == otherDice.getFaceValue())  
            return true;  
        else  
            return false;  
    }  
}
```

<<Java Class>>

 **SimpleDice4**

(default package)

 MAX: int faceValue: int SimpleDice4(int) SimpleDice4() roll():int getFaceValue():int setFaceValue(int):void toString():String equals(Object):boolean



equals()

Is it correct?

```
public boolean equals(Object o) {  
    Dice otherDice = (Dice) o;  
    if (this.getFaceValue() == otherDice.getFaceValue())  
        return true;  
    else  
        return false;  
}
```

```
public boolean equals(Object o) {  
    Dice otherDice = (Dice) o;  
    return this.getFaceValue() ==  
        otherDice.getFaceValue()  
}
```

```
public boolean equals(Object o) {  
    return this.getFaceValue() ==  
        ((Dice) o).getFaceValue()  
}
```



Does equals() work as expect?

- › Must be conformed with specification.
- › The equals method implements an equivalence relation on non-null object references:
 - It is **reflexive**: for any non-null reference value `x`, `x.equals(x)` should return true.
 - It is **symmetric**: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
 - It is **transitive**: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
 - It is **consistent**: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.
 - For any non-null reference value `x`, `x.equals(null)` should return false.



Testing x.equals(y)

	x	y	Expected result
reflexive	Non-null	-	true
symmetric	Non-null	Non-null	x.equals(y) == y.equals(x)
transitive	Non-null	Non-null	
consistent	Non-null		
x.equals(null)			false



Passing method arguments

- › Primitive type data is passed to a method “by value” (copy), while non-primitive type data is passed to a method “by reference”.
 - Pass by value: there is **no** change in the passing variable.
 - Pass by reference: the change in method affects the value of the passing variable.

Code

```
import java.awt.Point;

public class PassingDataToMethod {

    public static void main(String[] args) {

        int v = 2;

        Point p = new Point(2,2);

        passByValueSetToTen(v);

        passByReferenceSetToTen(p);

        System.out.println("v="+v);

        System.out.println(p.toString());

    }
```

```
public static void passByValueSetToTen(int a) {

    a = 10;

}

public static void passByReferenceSetToTen(Point a) {

    a.x = 10; a.y = 10;

}

}
```

Result

v=2

java.awt.Point[x=10,y=10]



Question

```
public class IdentifyMyParts {  
    public static int x = 7;  
    public int y = 3;  
}
```

What are the class variables?

What are the instance variables?



What is the output?

```
IdentifyMyParts a = new IdentifyMyParts();  
IdentifyMyParts b = new IdentifyMyParts();  
a.y = 5; b.y = 6; a.x = 1; b.x = 2;  
System.out.println("a.y = " + a.y);  
System.out.println("b.y = " + b.y);  
System.out.println("a.x = " + a.x);  
System.out.println("b.x = " + b.x);  
System.out.println("IdentifyMyParts.x = " +  
                    IdentifyMyParts.x);
```