

# OBJECT-ORIENTED PROGRAMMING USING JAVA

*USING AND DESIGNING OBJECTS*

QUAN THAI HA

HUS

FEBRUARY 13, 2024



## 1 Using Objects

- Java API
- Access Modifiers
- Class Methods and Instance Methods
- Constructors
- Overloading Methods
- Many Classes in Java API

## 2 Designing Objects

## 3 Packages

## 4 References

## 1 Using Objects

### ■ Java API

- Access Modifiers
- Class Methods and Instance Methods
- Constructors
- Overloading Methods
- Many Classes in Java API

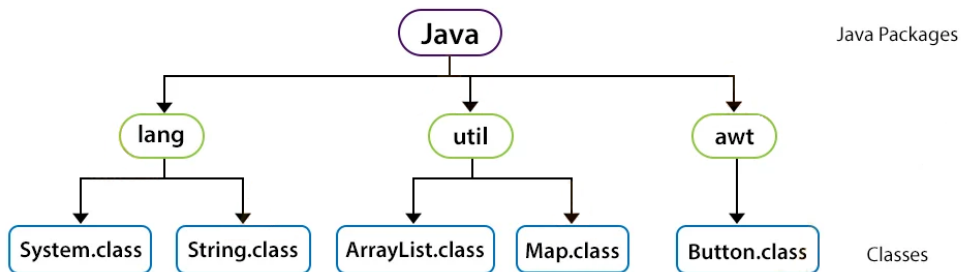
## 2 Designing Objects

## 3 Packages

## 4 References

- API stands for Application Programming Interface. Where you find service classes.

## Built-in Packages in Java



## ■ Java API Specification

► <https://docs.oracle.com/en/java/javase/19/docs/api/index.html>

OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
PREVIEW
NEW
DEPRECATED
INDEX
HELP

Java SE 19 & JDK 19

SEARCH

### Java® Platform, Standard Edition & Java Development Kit Version 19 API Specification

This document is divided into two sections:

**Java SE**

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with **java**.

**JDK**

The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform. These APIs are in modules whose names start with **jdk**.

All Modules	Java SE	JDK	Other Modules
Module	Description		
<b>java.base</b>	Defines the foundational APIs of the Java SE Platform.		
<b>java.compiler</b>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.		
<b>java.datatransfer</b>	Defines the API for transferring data between and within applications.		
<b>java.desktop</b>	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.		
<b>java.instrument</b>	Defines services that allow agents to instrument programs running on the JVM.		
<b>java.logging</b>	Defines the Java Logging API.		

- API documentation

- ▶ <https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Scanner.html>

- For reading input

- ▶ `import java.util.Scanner`

```
next()  
nextDouble()  
nextInt()  
nextLine()  
...
```

```
hasNext()  
hasNextDouble()  
hasNextInt()  
hasNextLine()  
...
```

Note Java naming convention  
Method names – **lowerCamelCase**

```
String  
next(String pattern)  
Returns the next token if it matches the pattern constructed from the specified string.  
BigDecimal  
nextBigDecimal()  
Scans the next token of the input as a BigDecimal.  
BigInteger  
nextBigInteger()  
Scans the next token of the input as a BigInteger.  
BigInteger  
nextBigInteger(int radix)  
Scans the next token of the input as a BigInteger.  
boolean  
nextBoolean()  
Scans the next token of the input into a boolean value and returns that value.  
byte  
nextByte()  
Scans the next token of the input as a byte.  
nextByte(int radix)  
Scans the next token of the input as a byte.  
nextDouble()  
Scans the next token of the input as a double.  
nextFloat()  
Scans the next token of the input as a float.  
nextInt()  
Scans the next token of the input as an int.  
nextInt(int radix)  
Scans the next token of the input as an int.  
nextLine()  
Advances this scanner past the current line and returns the input that was skipped.
```



```

1 import java.util.Scanner;

3 public class TestScanner {
    public static void main(String[] args) {
4         Scanner sc = new Scanner(System.in);
        // Comparing nextLine() and next()
5         System.out.print("Enter name1: ");
        String name1 = sc.nextLine();
6         System.out.println("name1 entered is '" + name1 + "'");
        System.out.print("Enter name2: ");
7         String name2 = sc.next();
        System.out.println("name2 entered is '" + name2 + "'");
8         sc.nextLine(); // to skip the rest of the line after the next() method
        // Using nextInt() and hasNextInt()
9         System.out.println("Enter integers, terminate with control-d:");
        int sum = 0;
10        while (sc.hasNextInt()) {
            int num = sc.nextInt();
11            System.out.println("Integer read: " + num);
            sum += num;
12        }
        System.out.println("Sum = " + sum);
13    }
14 }

```

## ■ API documentation

► <https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/String.html>

## ■ For using strings

► `import java.lang.String` (optional)

<code>String</code>	<code>toUpperCase(Locale locale)</code>	Converts all of the characters in this <code>String</code> to upper case using the rules of the given <code>Locale</code> .
<code>&lt;R&gt; R</code>	<code>transform(Function&lt;? super String,? extends R&gt; f)</code>	This method allows the application of a function to this string.
<code>String</code>	<code>translateEscapes()</code>	Returns a string whose value is this string, with escape sequences translated as if in a string literal.
<code>String</code>	<code>trim()</code>	Returns a string whose value is this string, with all leading and trailing space removed, where space is defined as any character whose codepoint is less than or equal to 'U+0020' (the space character).
<code>static String</code>	<code>valueOf(boolean b)</code>	Returns the string representation of the <code>boolean</code> argument.
<code>static String</code>	<code>valueOf(char c)</code>	Returns the string representation of the <code>char</code> argument.
<code>static String</code>	<code>valueOf(char[] data)</code>	Returns the string representation of the <code>char</code> array argument.
<code>static String</code>	<code>valueOf(char[] data, int offset, int count)</code>	Returns the string representation of a specific subarray of the <code>char</code> array argument.
<code>static String</code>	<code>valueOf(double d)</code>	Returns the string representation of the <code>double</code> argument.





```
import java.lang.String; (optional)

2
public class TestString {
4     public static void main(String[] args) {
        String text = new String("I'm studying OOP."); // or String text = "I'm studying OOP.";
6         // We will explain the difference later.

8         System.out.println("text: " + text);
        System.out.println("text.length() = " + text.length());
10        System.out.println("text.charAt(5) = " + text.charAt(5)); // t
        System.out.println("text.substring(5, 8) = " + text.substring(5, 8)); // tud
12        System.out.println("text.indexOf(\"in\") = " + text.indexOf("in"));

14        String newText = text + "How about you?";
        newText = newText.toUpperCase();
16        System.out.println("newText: " + newText);
        if (text.equals(newText)) {
18            System.out.println("text and newText are equal.");
        } else {
20            System.out.println("text and newText are not equal.");
        }
22    }
}
```

- As strings are objects, do not use `==` if you want to check if two strings contain the same text.
- Use the `equals()` method provided in the `String` class instead. (Read more details about `equals()` in Java documentation).



```
1 Scanner sc = new Scanner(System.in);
  System.out.println("Enter 2 identical strings:");
3 String str1 = sc.nextLine();
  String str2 = sc.nextLine();
5 System.out.println(str1 == str2);
  System.out.println(str1.equals(str2));
7
  // Enter 2 identical strings:
9  // Hello world!
  // Hello world!
11
  // (What will be printed?)
```

## ■ API documentation

- ▶ <https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/Math.html>

## ■ For using math functions

- ▶ `import java.lang.Math` (optional)

Fields		
Modifier and Type	Field	Description
<code>static final double</code>	<code>E</code>	The double value that is closer than any other to $e$ , the base of the natural logarithms.
<code>static final double</code>	<code>PI</code>	The double value that is closer than any other to $\pi$ ( $n$ ), the ratio of the circumference of a circle to its diameter.
<code>static final double</code>	<code>TAU</code>	The double value that is closer than any other to $\tau$ ( $\tau$ ), the ratio of the circumference of a circle to its radius.

## Method Summary

All Methods			Static Methods	Concrete Methods
Modifier and Type	Method		Description	
<code>static double</code>	<code>abs(double a)</code>		Returns the absolute value of a <code>double</code> value.	
<code>static float</code>	<code>abs(float a)</code>		Returns the absolute value of a <code>float</code> value.	
<code>static int</code>	<code>abs(int a)</code>		Returns the absolute value of an <code>int</code> value.	
<code>static long</code>	<code>abs(long a)</code>		Returns the absolute value of a <code>long</code> value.	



```
import java.util.Scanner;
2 import java.lang.Math; (optional)

4 public class TestMath {
    public static void main(String[] args) {
6         Scanner sc = new Scanner(System.in);
        System.out.print("Enter 3 values: ");
8         double num1 = sc.nextDouble();
        double num2 = sc.nextDouble();
10        double num3 = sc.nextDouble();

12        System.out.printf("pow(%.2f,%.2f) = %.3f\n", num1, num2, Math.pow(num1, num2));
        System.out.println("Largest = " + Math.max(Math.max(num1, num2), num3));
14        System.out.println("Generating 5 random values:");

16        for (int i = 0; i < 5; i++) {
            System.out.println(Math.random());
18        }
20    }
```

## 1 Using Objects

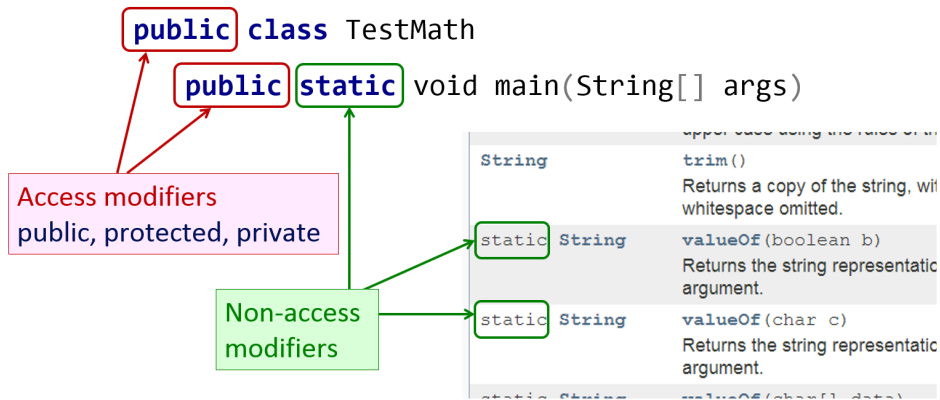
- Java API
- **Access Modifiers**
- Class Methods and Instance Methods
- Constructors
- Overloading Methods
- Many Classes in Java API

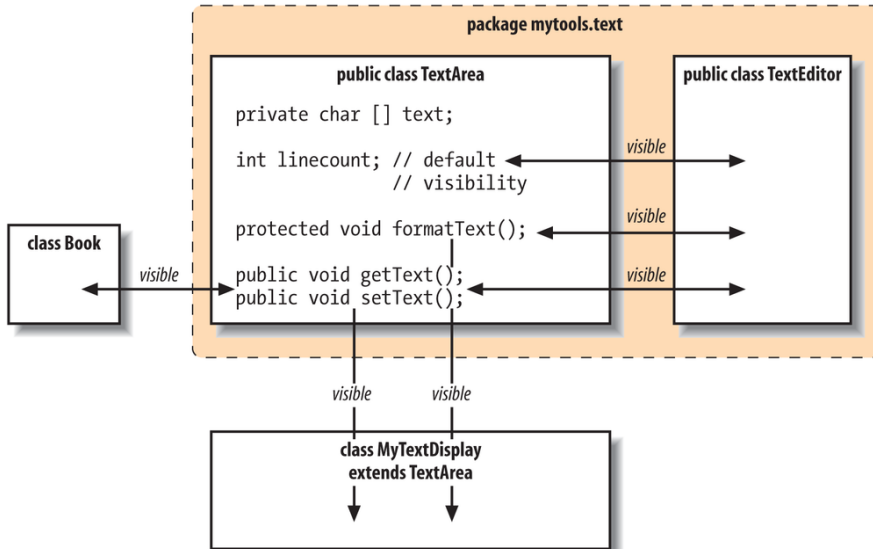
## 2 Designing Objects

## 3 Packages

## 4 References

- Modifiers: keywords added to specify the way a class/attribute/method works.





## 1 Using Objects

- Java API
- Access Modifiers
- **Class Methods and Instance Methods**
- Constructors
- Overloading Methods
- Many Classes in Java API

## 2 Designing Objects

## 3 Packages

## 4 References



String	translateEscapes()
String	trim()

static String	valueOf(boolean b)
static String	valueOf(char c)
static String	valueOf(char[] data)

- All methods in the Math class are static methods.
- All methods in the Scanner class are non-static methods.
- The String class comprises a mix of static and non-static methods.

static double	max(double a, double b)
static float	max(float a, float b)
static int	max(int a, int b)
static long	max(long a, long b)
static double	min(double a, double b)
static float	min(float a, float b)
static int	min(int a, int b)

boolean	hasNext()
boolean	hasNext(String pattern)
boolean	hasNext(Pattern pattern)
boolean	hasNextBigDecimal()
boolean	hasNextBigInteger()

String	translateEscapes()
String	trim()
static String	valueOf(boolean b)
static String	valueOf(char c)
static String	valueOf(char[] data)

static double	max(double a, double b)
static float	max(float a, float b)
static int	max(int a, int b)
static long	max(long a, long b)
static double	min(double a, double b)
static float	min(float a, float b)
static int	min(int a, int b)

- A static method (preferably called a class method) means that no object (instance) of the class is needed to use the method.
- A non-static method (preferably called an instance method) means that the method must be applied to an object (instance) of that class.

boolean	hasNext()
boolean	hasNext(String pattern)
boolean	hasNext(Pattern pattern)
boolean	hasNextBigDecimal()
boolean	hasNextBigInteger()



```
public class Cone {
2   public static void main(String[] args) {
    // To call static method, preceding method with the class name.
4   double vol = Cone.volumeCone(radius , height);

6   /*
    * Optional to precede method with the class name
    * if the method is defined in the class it is called.
    *
10   * Alternatively:
    * double vol = volumeCone(radius , height);
12   */
    }

14   public static double volumeCone(double radius , double hight) {
16       return Math.PI * radius * radius * hight / 3.0;
    }

18 }
```



```
1 public class Cone {
2     public static void main(String[] args) {
3
4         // An instance method must be applied to an instance (object) of a class.
5         Cone newCone = new Cone(1.0 , 2.0);
6         double volume = newCone.volumeCone(radius , height);
7
8         /*
9          * Calling an instance method is referred to as passing a message to an instance (object).
10        */
11    }
12
13    public double volumeCone(double radius , double hight) {
14        return Math.PI * radius * radius * hight / 3.0;
15    }
16 }
```

## 1 Using Objects

- Java API
- Access Modifiers
- Class Methods and Instance Methods
- **Constructors**
- Overloading Methods
- Many Classes in Java API

## 2 Designing Objects

## 3 Packages

## 4 References

- When a class (eg: **String**, **Scanner**) provides instance methods, it expects instances (objects) to be created from that class.
- This requires a special method called a **constructor**. The keyword **new** is used to invoke the constructor.

Constructor and Description
<code>Scanner(File source)</code> Constructs a new <code>Scanner</code> that produces values scanned from the specified file.

```
Scanner sc = new Scanner(System.in);
```

Note the keyword **new**

Constructor and Description
<code>String()</code> Initializes a newly created <code>String</code> object so that it represents an empty character sequence.
<code>String(String original)</code> Initializes a newly created <code>String</code> object so that it represents the same sequence of characters

```
String str1 = new String();  
String str2 = new String("To be or not to be?");
```

## 1 Using Objects

- Java API
- Access Modifiers
- Class Methods and Instance Methods
- Constructors
- **Overloading Methods**
- Many Classes in Java API

## 2 Designing Objects

## 3 Packages

## 4 References

- Some methods have identical names, but with different parameters. This is called **overloading**.

## Overloaded Methods

Modifier and Type	Method
Description	
static double	<b>abs</b> (double a)
Returns the absolute value of a <b>double</b> value.	
static float	<b>abs</b> (float a)
Returns the absolute value of a <b>float</b> value.	
static int	<b>abs</b> (int a)
Returns the absolute value of an <b>int</b> value.	

## Overloaded Constructors

Constructors
Constructor
Description
<b>String()</b>
Initializes a newly created <b>String</b> object so that it represents an empty character sequence.
<b>String(byte[] bytes)</b>
Constructs a new <b>String</b> by decoding the specified array of bytes using the <b>default charset</b> .



- Without overloading, different named methods would have to be provided:



```
absDouble(double a)
2 absFloat(float a)
absInt(int a)
4 absLong(long a)
```

- With overloading, all these related methods have the same name.



```
abs(double a)
2 abs(float a)
abs(int a)
4 abs(long a)
```

## 1 Using Objects

- Java API
- Access Modifiers
- Class Methods and Instance Methods
- Constructors
- Overloading Methods
- Many Classes in Java API

## 2 Designing Objects

## 3 Packages

## 4 References



```
import java.text.DecimalFormat;
2
public class TestDecimalFormat {
4     public static void main(String[] args) {
        DecimalFormat df1 = new DecimalFormat("0.000");
        DecimalFormat df2 = new DecimalFormat("#.###");
        DecimalFormat df3 = new DecimalFormat("0.00%");
        System.out.println("PI = " + df1.format(Math.PI));
        System.out.println("12.3 formatted with \"0.000\" = " + df1.format(12.3));
        System.out.println("12.3 formatted with \"#.###\" = " + df2.format(12.3));
        System.out.println("12.3 formatted with \"0.00%\" = " + df3.format(12.3));
12     }
}
```

## Command window



```
1  PI = 3.142
   12.3 formatted with "0.000" = 12.300
3  12.3 formatted with "#.###" = 12.3
   12.3 formatted with "0.00%" = 1230.00%
```



```
import java.util.Random;
2
public class TestRandom {
4     public static void main(String[] args) {
        // To generate a random integer in [51, 70]
        // using Math.random() and Random's nextInt()
6         int num1 = (int)(Math.random() * 20) + 51;
        System.out.println("num1 = " + num1);
8
10        Random random = new Random();
        int num2 = random.nextInt(20) + 51;
12        System.out.println("num2 = " + num2);
    }
14 }
```

## Command window



```
num1 = 51
2 num2 = 68
```

## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- Overloading method
- Getters and Setters
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References

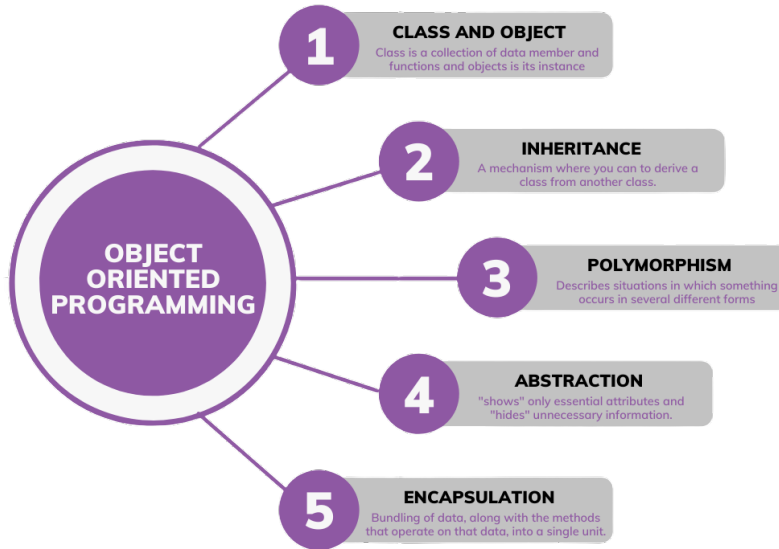
## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- Overloading method
- Getters and Setters
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References

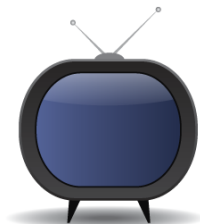


- If we think of a real-world object, such as a television, it will have several features and properties:
  - ▶ We do not have to open the case to use it.
  - ▶ We have some controls to use it (buttons on the box, or a remote control).
  - ▶ We can still understand the concept of a television, even if it is connected to a DVD player.
  - ▶ It is complete when we purchase it, with any external requirements well documented.



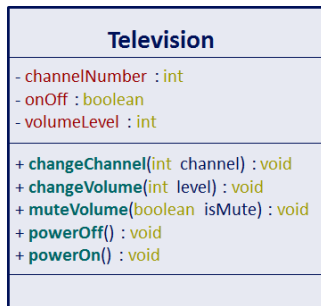


- Classes allow us a way to represent complex structures within a programming language. They have two components:
  - ▶ **State** - (or data) are the values that the object has.
  - ▶ **Methods** - (or behaviour) are the ways in which the object can interact with its data, the actions.



Example **States**

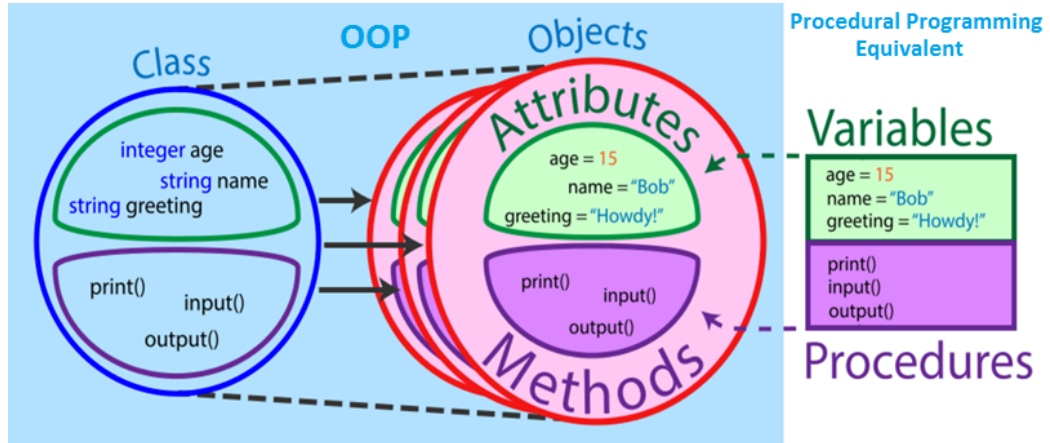
Example **Methods**



Unified Modelling Language (UML) representation of the Television class

- An **instance of a class is called an object.**
- You could think of a class as the description of a concept, and an object as the realization of this description to create an independent distinguishable entity.
- For example, in the case of the television, the class is the set of plans (or blueprints) for a generic television, whereas a television object is the realization of these plans into a real-world physical television.
- There would be one set of plans (the class), but there could be thousands of real-world televisions (objects).
- Objects are concrete (a real-world object, a file on a computer).

- **Class:** a template – or a blueprint – that defines the form of an object.
  - ▶ Data (**variables, attributes**)
  - ▶ Code (**behaviours, operations, methods**)
- **Object:** Objects are used to store data, act on that data and interact with other objects.
  - ▶ Identity
  - ▶ Type
  - ▶ Internal state, behaviours
  - ▶ One or more references must point to every valid object
- A class is like a **type definition**. **No data is allocated** until an object is created from the class.
- The creation of an object is called instantiation. The created object is called an **instance**.
- No limit to the number of objects that can be created from a class.
- Each object is independent. Changing one object doesn't change the others.





```
package com.vehicle;
2
public class Car {
4     // Attributes
    private String company;
6     private int speed;

8     // Constructors
    public Car(String company, int speed) {
10         this.company = company;
        this.speed = speed;
12     }

14     // Methods
    public void getSpeed() {
16         System.out.println(company + "car's speed is" + speed + "Km/hr.");
    }

18
    public static void main(String[] args) {
20         Car car1 = new Car("Honda", 100);
        Car car2 = new Car("Jeep", 500);
22         Car car3 = new Car("BMW", 800);
    }
24 }
```

## Car class

```
class Car {  
    String company;  
    int speed;  
  
    void getSpeed() {  
        System.out.println(company +  
            "car's speed is " + speed +  
            "Km/hr");  
    }  
}
```

## Multiple Objects



Company = Honda  
Speed = 100  
Honda car's speed is 100  
Km/hr



Company = Jeep  
Speed = 500  
Jeep car's speed is 500  
Km/hr



Company = BMW  
Speed = 800  
BMW car's speed is 800  
Km/hr



```
package domain.internet.packagename
2
access-modifier class DataType
4 // attributes
  access-modifier data-type attribute1
  access-modifier data-type attribute2
6
8 // constructors
  access-modifier DataType(attribute1, ..., attribute2)
10
12 // methods
  access-modifier data-type doOperation1(parameters)
  access-modifier data-type doOperation2(parameters)
14
16 // Create Objects
  DataType instance1 = new DataType(attribute1, ..., attribute2);
18  DataType instance2 = new DataType(attribute1, ..., attribute2);
```

- The creation of an object is made with the keyword **new** which creates a new instance of the specified class, and allocates it in memory (Heap in Java).
  - ▶ **new calls the constructor method of the class** (a method without return type and with the same name of the Class).
  - ▶ **new returns a reference** to the portion of memory containing the created object.



```
DataType ref = new DataType(parameters);
```

```
2
```

```
/*
```

```
4 * - The operator new invokes a constructor of DataType class.
```

```
* - A instance (object) of DataType class is allocated in Heap memory
```

```
6 * (in Java).
```

```
* - The references ref is allocated in Stack memory,
```

```
8 * reference to the portion of memory (Heap) containing the created object.
```

```
*/
```

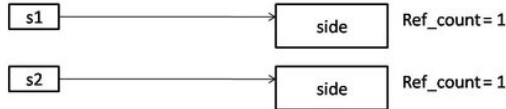


- It is no longer a developer concern. Java supports **Automatic Garbage Collection** (an automatic way for de-allocating unreferenced objects).

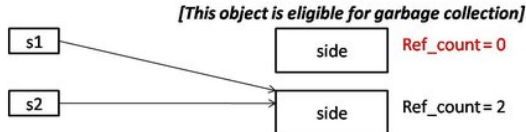
**Square s1 = new Square();**



**Square s2 = new Square();**



**s1 = s2;**





```
1  /**
   * Class BankAccount.
   */
3  */
   public class BankAccount {
5      // Attributes
       private int accountNumber;
7      private double balance;

9      // Constructors
       // Name must be identical to class name. No return type.
11     public BankAccount() {
           // By default, numeric attributes are initialised to 0
13     }

15     public BankAccount(int accountNumber, double balance) {
           // Initialize attributes with user provided values
17         this.accountNumber = accountNumber;
           this.balance = balance;
19     }
```



```
1  // Getter methods
   public int getAccountNumber() {
3      return accountNumber;
   }

5
   public double getBalance() {
7      return balance;
   }

9
   // Setter methods
11  public void setAccountNumber(int accountNumber) {
      this.accountNumber = accountNumber;
13  }

15  public void setBalance(double balance) {
      this.balance = balance;
17  }
```



```
1  // User methods
   public boolean withdraw(double amount) {
3      if (balance < amount)
           return false;
5      balance -= amount;
           return true;
7  }

9  public void deposit(double amount) {
       if (amount <= 0)
11         return;
           balance += amount;
13     }

15     public void print() {
           System.out.println("Account number: " + getAccountNumber());
17         System.out.printf("Balance: $%.2f\n", getBalance());
           }
19 }
```



- **Attributes** are also called Member Data, or Fields (in Java API documentation).
- **Behaviours** (or Member Behaviours) are also called Methods (in Java API documentation).
- Attributes and members can have different level of **accessibilities/visibilities**.
- Each class has one or more **constructors**:
  - ▶ To initialize the data when creating an instance of an class.
  - ▶ **Default constructor** has no parameter and is automatically generated by compiler if class designer does not provide any constructor.
  - ▶ Constructors can be overloaded.

- Encapsulation is defined as the mechanism wrapping together code and data (data is encapsulated inside a shield of code).
- Another way to think about encapsulation is a protective shield that prevents the data from being accessed by code outside this shield.



```
1 public class Car {  
    public String color; // using public access modifier  
3  
    public void setColor(String color) {  
5        this.color = color;  
    }  
7 }  
  
9 Car car = new Car();  
   car.color = "red"; // Works but unsafe!
```

## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- **Encapsulation and visibility**
- Constructors
- Overloading method
- Getters and Setters
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

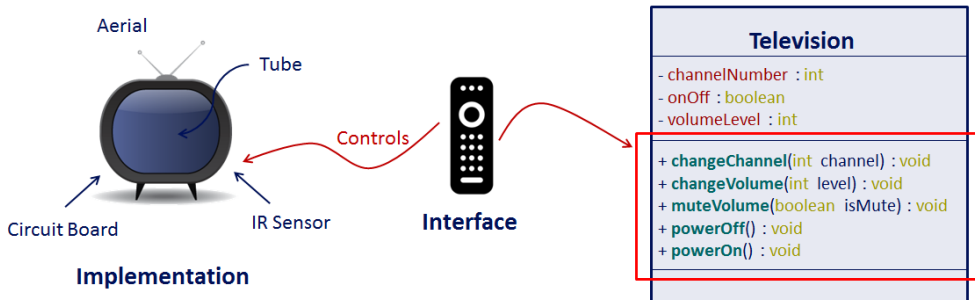
## 4 References

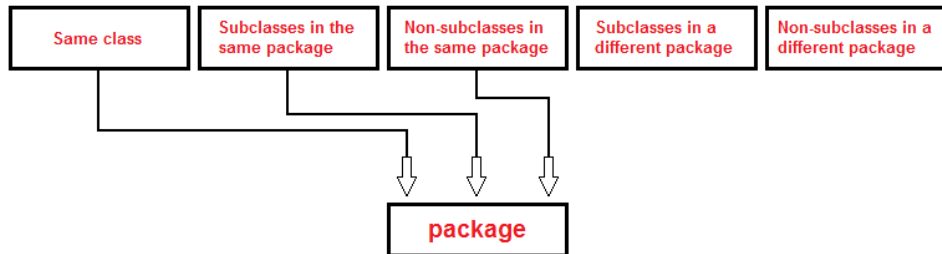
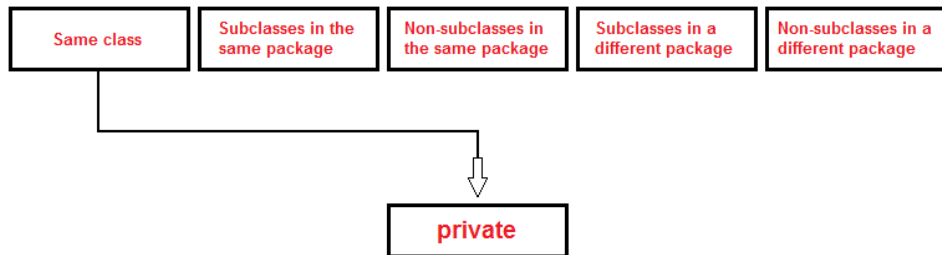


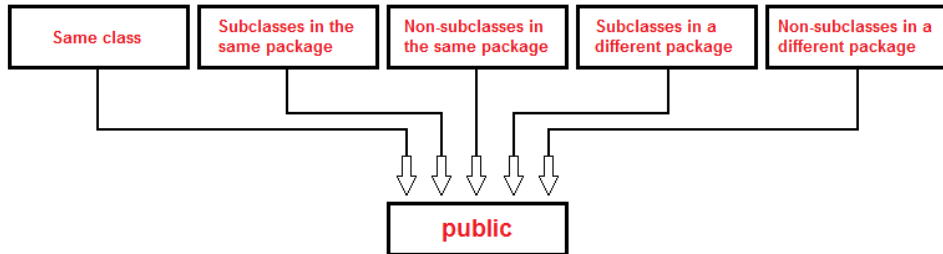
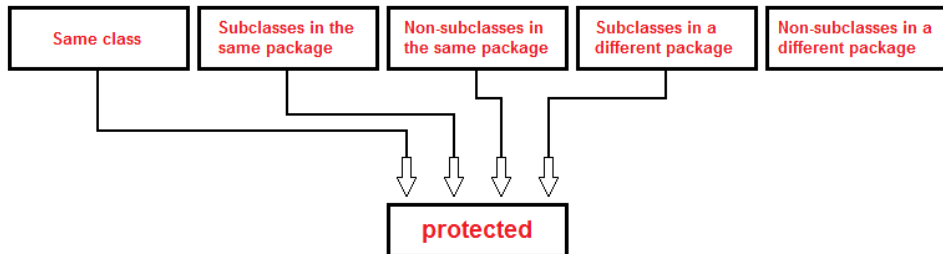
```
1 public class Car {
2     private String color; // using private access modifier
3
4     public void setColor(String color) {
5         this.color = color;
6     }
7 }
8
9 public class App {
10     public static void main(String[] args) {
11         Car car = new Car();
12         car.color = "red"; // Compiler error
13         car.setColor("red"); // Works, Safe!
14     }
15 }
```



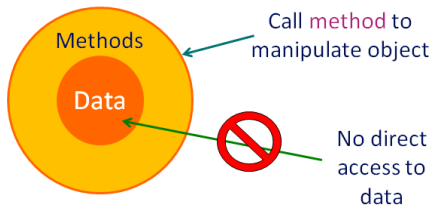
- The object-oriented paradigm encourages encapsulation.
- Encapsulation is used to hide the mechanics of the object, allowing the actual implementation of the object to be hidden, so that we don't need to understand how the object works.
- There is a subset of functionality that the user is allowed to call, termed the interface.
- The full implementation of a class is the sum of the public interface plus the private implementation.

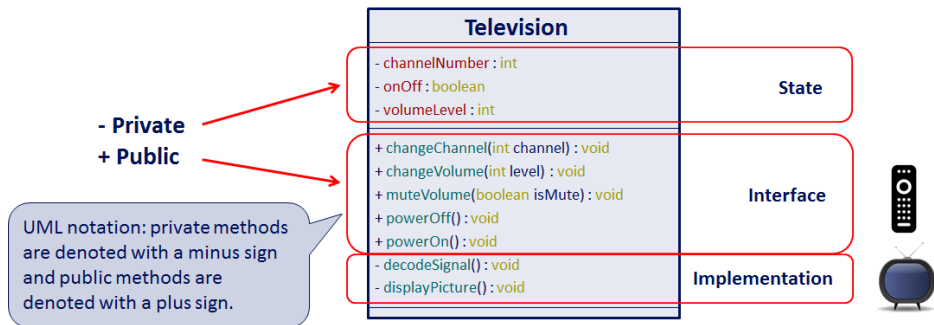






- **Attributes** are usually **private**.
  - ▶ Information hiding, to shield data of an object from outside view.
  - ▶ We provide **public methods** for user to access the attributes through the public methods.
- **Methods** are usually **public**
  - ▶ so that they are available for users.
    - Imagine that the methods in **String** class and **Math** class are private instead, then we cannot even use them!
  - ▶ If the methods are to be used internally in the service class itself and not for users, then the methods should be declared private instead.





- The **private** methods would be methods written that are part of the inner workings of the television, but need not be understood by the user.
- For example, the user would need to call the **powerOn()** method but the private **displayPicture()** method would also be called, but internally as required, not directly by the user. This method is therefore not added to the interface, but hidden internally in the implementation by using the **private** keyword.

## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- **Constructors**
- Overloading method
- Getters and Setters
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References

- A constructor is a special method that is used to initialize objects. The constructor contain operations we want to execute as soon as objects are created (**attributes initialization!**). The constructor is called when an object of a class is created.
- **Overloading** of constructors is often used.
- If a constructor is not defined within a class, a default one (with no parameters) is defined. If a constructor is defined, the default one is disabled!



```
1 public class Car {  
    private String brand;  
    private String color;  
  
5    // Defined constructor  
    public Car() {  
6        this.brand = "Fiat";  
7        this.color = "Punto";  
9    }  
  
11   // Overloaded constructor  
    public Car(String brand, String color) {  
13        this.brand = brand;  
14        this.color = color;  
15    }  
  
17    ...  
  
19    /*  
    * Constructors can be automatically generated  
21    * in IDEs. In IntelliJ, Code -> Generate ...  
    */  
23 }
```



```
1 public class Car {  
    private String brand;  
3    private String color;  
  
5    // If the constructor is not defined, compiler will create  
    // a default constructor (constructor without parameters).  
7  
    public void setColor(String color) {  
9        this.color = color;  
    }  
11 }  
  
13 public class App {  
    public static void main(String[] args) {  
15        // Works with default constructor!  
        // Possibly unsafe: attributes not initialized.  
17        Car car = new Car();  
    }  
19 }
```





```
1 public class Car {  
    private String brand;  
3    private String color;  
  
5    // Constructor  
    public Car(String brand, String color) {  
6        this.brand = brand;  
7        this.color = color;  
9    }  
  
11   // Setter method  
    public void setColor(String color) {  
13        this.color = color;  
15    }  
  
17 Car car = new Car();           // Error! Default Constructor missing!  
    Car car = new Car("Audi", "Red"); // Works with defined constructor!
```



```
public class Car {  
2   private String brand;  
   private String color;  
4  
   public Car() {  
6       this.brand = "Audi";  
       this.color = "Red";  
8   }  
  
   public Car(String brand, String color) {  
10      this.brand = brand;  
12      this.color = color;  
   }  
14  
   /* ... */  
16 }  
  
18 Car car = new Car();           // Works with defined constructor!  
   Car car = new Car("Audi", "Red"); // Works with defined constructor!
```

- Besides constructors, there are two other types of special methods that be referred to as **accessors** and **mutators**.
- An accessor is a method that accesses (retrieves) the value of an object's attribute.
  - ▶ E.g.: **getAccountNumber()**, **getBalance()**.
  - ▶ Its return type must match the type of the attribute it retrieves.
- A mutator is a method that mutates (modifies) the value of an object's attribute.
  - ▶ E.g.: **withdraw()**, **deposit()**.
  - ▶ Its return type is usually. **void**, and it usually takes in some argument to modify the value of an attribute.

- As a (service) class designer, you decide the following:
  - ▶ What attributes you want the class to have.
  - ▶ What methods you want to provide for the class so that users may find them useful.
  - ▶ For example, the `print()` method is provided for `BankAccount` as the designer feels that it might be useful. Or, add a `transfer()` method to transfer money between 2 accounts?
  - ▶ As in any design undertaking, there are no hard and fast rules. One approach is to study the classes in the API documentation to learn how others designed the classes, and google to explore.
- The classes that contain the `main()` method are called client classes.
  - ▶ Note that there is no `main()` method in `BankAccount` class because it is a service class, not a client class (application program). You cannot execute `BankAccount`.
  - ▶ In general, the service class and the client class may be put into a single `.java` program, mostly for quick testing. (However, there can only be 1 public class in such a program, and the public class name must be identical to the program name.)
  - ▶ We will write `1 class per .java` program here (most of the time) to avoid confusion.



```
1 public class TestBankAccount {  
    public static void main(String[] args) {  
3         BankAccount bankAccount1 = new BankAccount();  
         BankAccount bankAccount2 = new BankAccount(1234, 321.70);  
5  
         System.out.println("Before transactions:");  
7         bankAccount1.print();  
         bankAccount2.print();  
9  
         bankAccount1.deposit(1000);  
11        bankAccount1.withdraw(200.50);  
         bankAccount2.withdraw(500.25);  
13  
         System.out.println();  
15        System.out.println("After transactions:");  
         bankAccount1.print();  
17        bankAccount2.print();  
    }  
19 }
```

## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- **Overloading method**
- Getters and Setters
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References

- Methods may have parameters.
- Inside a Java class, methods with the same name but different overall signatures are allowed.
- A signature is made by:
  - ▶ Method name
  - ▶ Ordered list of parameters types
- The method whose parameters types matches, is selected to be executed.



```
1 public class Foo {  
    public static void main(String[] args) {  
3        Foo foo = new Foo();  
        System.out.println(foo.doit(5, "Foo"));  
5        // 8  
        System.out.println(foo.doit("Foo", 5));  
7        // 15  
    }  
9  
    // Overloading methods  
11    public int doit(int n, String s) {  
        return n + s.length();  
13    }  
  
15    public int doit(String s, int n) {  
        return n * s.length();  
17    }  
}
```

## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- Overloading method
- **Getters and Setters**
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References



- Since **attributes** are usually **encapsulated**, methods for reading and writing them are useful. These methods are called **getters** and **setters**.
- It is worth noting that, for reducing the number of errors, using the same name for method parameters and class attributes is a good practice!



```
public class Car {  
2   private String color;  
   ...  
  
4  
   // Getter  
6   public String getColor() {  
       return color;  
8   }  
  
10  // Setter  
11  public void setColor(String color) {  
12      this.color = color;  
13  }  
14  
   ...  
16  
   /*  
18    * Getters and Setters can be  
19    * automatically generated in IDEs.  
20    * In IntelliJ, Code -> Generate ...  
21    */  
22 }
```

Why use getters and setters/accessors?

- Encapsulation of behaviour associated with getting or setting the property - this allows additional functionality (like validation) to be added more easily later.
- Hiding the internal representation of the property while exposing a property using an alternative representation.
- Insulating your public interface from change - allowing the public interface to remain constant while the implementation changes without affecting existing consumers.
- Controlling the lifetime and memory management (disposal) semantics of the property - particularly important in non-managed memory environments (like C++ or Objective-C).
- Providing a debugging interception point for when a property changes at runtime - debugging when and where a property changed to a particular value can be quite difficult without this in some languages.

Why use getters and setters/accessors?

- Improved interoperability with libraries that are designed to operate against property getter/setters - Mocking, Serialization, and WPF come to mind.
- Allowing inheritors to change the semantics of how the property behaves and is exposed by overriding the getter/setter methods.
- Allowing the getter/setter to be passed around as lambda expressions rather than values.
- Getters and setters can allow different access levels - for example the get may be public, but the set could be protected.
- Because from now when you realize that your setter needs to do more than just set the value, you'll also realize that the property has been used directly in many other classes.

- It is handy to obtain a textual representation from objects.
- In order to provide objects with this feature the method `toString()` have to be overridden (the default implementation resides in the `Object` class).
- *Can be automatically generated in IDEs. In IntelliJ Code -> Generate...*



```

1 public class Point {
2     private int x;
3     private int y;
4
5     public Point(int x, int y) {
6         this.x = x;
7         this.y = y;
8     }
9
10    public static void main(String[] args) {
11        Point point = new Point(2, 3);
12        System.out.println(point);
13        System.out.println(System.identityHashCode(point));
14    }
15 }

```

## Command window

```

1 // Result without overriding toString() method.
2 Point@e9e54c2
3 245257410

```



```

1 public class Point {
2     private int x;
3     private int y;
4
5     /* ... */
6
7     @Override
8     public String toString() {
9         return "Point[" + "x=" + x + ", y=" + y + "]";
10    }
11
12    public static void main(String[] args) {
13        System.out.println(new Point(2, 3));
14    }
15 }

```

## Command window



```

1 // Result with overriding toString() method.
  Point[x=2, y=3]

```

## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- Overloading method
- Getters and Setters
- **Static attributes and methods**
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References

- A class comprises 2 types of members: **attributes** (data members) and **methods** (behaviour members).
- Java provides the modifier **static** to indicate if the member is a class member or an instance member.

	Attribute	Method
static	Class attribute	Class method
default	Instance attribute	Instance method

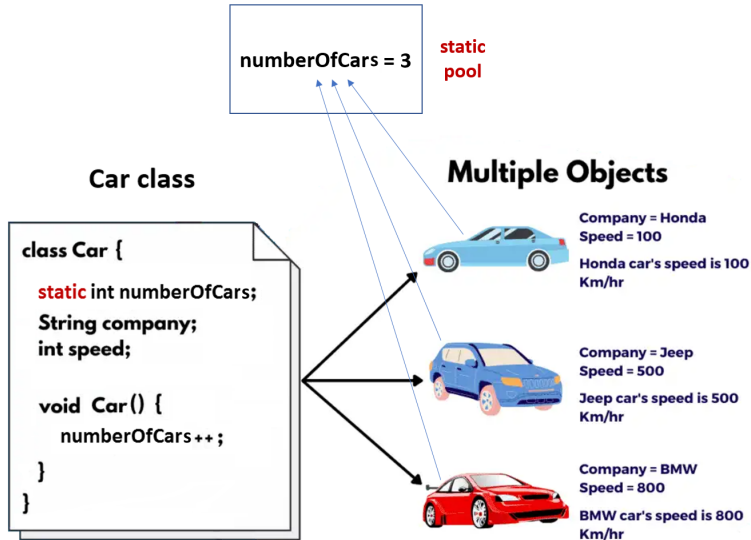
- Static attributes and methods are common to all instances of an object.
- They exist even when no object has been instantiated!
- Widely used for defining traditional functions inside OO software. Classes, in this case, are used just as containers.
- See **Math**, **Arrays**, **Collections** classes.
- Access:

**ClassName.attribute|method**  
or  
**reference.attribute|method**





```
public class Car {  
2   static int numberOfCars = 0;  
  
4   static int getCars() {  
    return numberOfCars;  
6   }  
  
8   public static void main(String[] args) {  
    // Access to static attributes  
10   int numCars = Car.numberOfCars;  
    double pi = Math.PI;  
12  
    // Access to static methods  
14   Car.getCars();  
    double cosValue = Math.cos(pi);  
16 }  
}
```



## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- Overloading method
- Getters and Setters
- Static attributes and methods
- **The keyword "this"**
- Wrapper classes

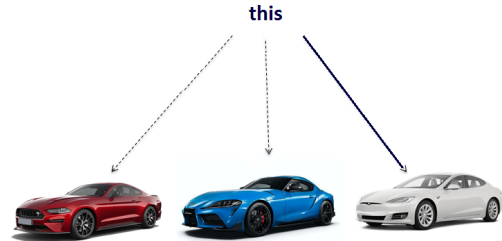
## 3 Packages

## 4 References

- A common confusion:
  - ▶ How does the method “know” which is the “object” it is currently communicating with? (Since there could be many objects created from that class.)
- Whenever a method is called,
  - ▶ a reference to the calling object is set automatically.
  - ▶ Given the name “**this**” in Java, meaning “this particular object”.
- All attributes/methods are then accessed implicitly through this reference.
- **this** represents a reference to the current object.
- It can be useful in methods to distinguish between instance attributes and local variables.
- Accessing attributes or methods of the same object do not require using **this** if there are no ambiguities.



```
1 public class Car {  
    private String color;  
3  
    public Car(String color) {  
6        this.color = color;  
    }  
7  
    public Car setColor(String color) {  
9        this.color = color;  
        // Return current object  
11       return this;  
    }  
13 }  
  
15 car1.setColor("Red");  
    car2.setColor("Blue");  
17 car3.setColor("White"); // calling object
```



- The “this” is optional for unambiguous case.



```
1 public String setColor(String newColor) {  
    color = newColor;  
3    // Or this.color = newColor;  
}
```

- “this” cannot be used in a static method. Why?



```
public static int getNumberOfCars() {  
2    return this.numberOfCars;  
}
```

The keyword "**this**" is used:

- 1. Refers to the current class variable**
- 2. Refers to the current class method**
- 3. Used in constructor chaining**
- 4. this as an argument in the method call**
- 5. this as an argument in the constructor call**
- 6. Using 'this' keyword to return current class instance**

- Dotted notations can be combined



```
1 System.out.println("Hello world!");
```

- **System** is a class in package java.lang.
- **out** is a (static) attribute of System referencing an object of class PrintStream (representing the standard output).
- **println()** is a method of PrintStream which prints a given string.



## 1 Using Objects

## 2 Designing Objects

- Objects and Classes
- Encapsulation and visibility
- Constructors
- Overloading method
- Getters and Setters
- Static attributes and methods
- The keyword "this"
- Wrapper classes

## 3 Packages

## 4 References

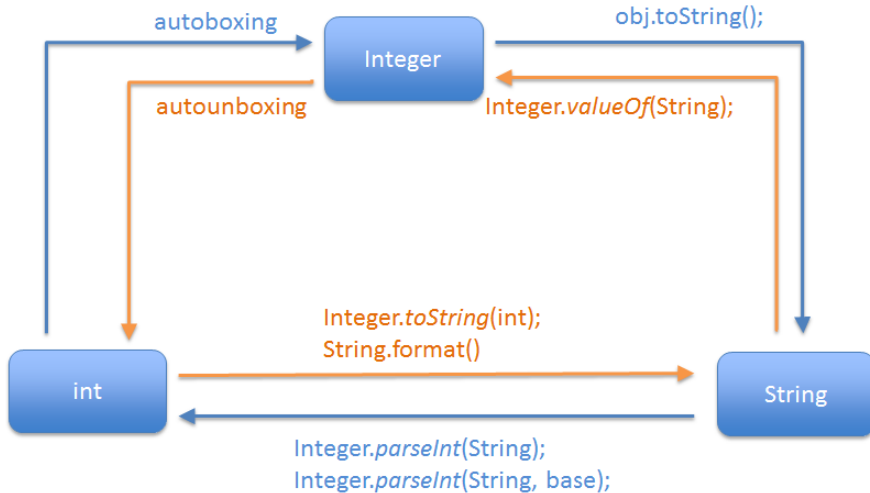
- In an ideal OO environment, only classes and objects should exist.
- For the sake of **performance**, Java also uses **primitive types** (**int**, **float**, etc.). Other languages, such as Python, do not support primitive types (everything is an object).
- **Wrapper classes are objectivized versions of primitive types.**
- Wrapper classes provide several methods for converting to and from primitive types, objectivized versions and strings, as well as other constants and useful methods.

- |           |                  |
|-----------|------------------|
| ■ boolean | <b>Boolean</b>   |
| ■ char    | <b>Character</b> |
| ■ byte    | <b>Byte</b>      |
| ■ short   | <b>Short</b>     |
| ■ int     | <b>Integer</b>   |
| ■ long    | <b>Long</b>      |
| ■ float   | <b>Float</b>     |
| ■ double  | <b>Double</b>    |
| ■ void    | <b>Void</b>      |

- In an ideal OO environment, only classes and objects should exist.
- For the sake of **performance**, Java also uses primitive types (int, float, etc.). Other languages, such as Python, do not support primitive types (everything is an object).
- **Wrapper classes are objectivized versions of primitive types.**
- Wrapper classes provide several methods for converting to and from primitive types, objectivized versions and strings, as well as other constants and useful methods.



```
public class MyInteger {  
2   private int n;  
  
4   public MyInteger(int n) {  
        this.n = n;  
6   }  
  
8   public int getN() {  
        return n;  
10  }  
  
12  public void setN(int n) {  
        this.n = n;  
14  }  
}
```



- **Auto boxing** is the automatic conversion from primitive types and their corresponding wrapper classes. The plain variable is boxed into an object.
  - For example, converting an **int** to an **Integer**, a **double** to a **Double**, and so on.
- **Unboxing**, on the contrary, is the automatic conversion from wrapper classes to their corresponding primitive types.



```
1 public class AutoboxingExample {  
    public static void main(String[] args) {  
3        // Auto boxing 2(int) -> Integer(2)  
        // Unboxing Integer(2) -> 2(int)  
5        int i = 2 + method(2);  
    }  
7  
    public static Integer method(Integer n) {  
9        System.out.println(n);  
        return n;  
11    }  
}
```

- 1 Using Objects
- 2 Designing Objects
- 3 Packages**
- 4 References

- Classes are better at modularizing code than functions.
- For the sake of additional modularization, packages are used.
- A package is a set of class definitions all stored within the same directory.
- Visibility rules apply to packages.
- The public interface of a package is the default portion of the classes contained in the package.
  - ▶ Minimize the number of classes, attributes, methods (of the package) visible from the outside.

- **package packagename;**
  - ▶ Package statement (first line of class file)
- **import packagename.ClassName;**
  - ▶ Import statements (after package statement)
  - ▶ If two packages contain a class with the same name, they cannot be both imported. If you need both classes you have to use one of them with its fully-qualified name.
- A package is identified by a name with a hierarchic structure (fully qualified name)
  - ▶ java.lang
  - ▶ java.util
- Conventions for defining unique names is based on **Internet naming but in reverse order (from general to specific concepts)**
  - ▶ org.oop.basics
- It's possible to use same class names in different packages without conflicts
  - ▶ **java.util.Date**
  - ▶ **java.sql.Date**

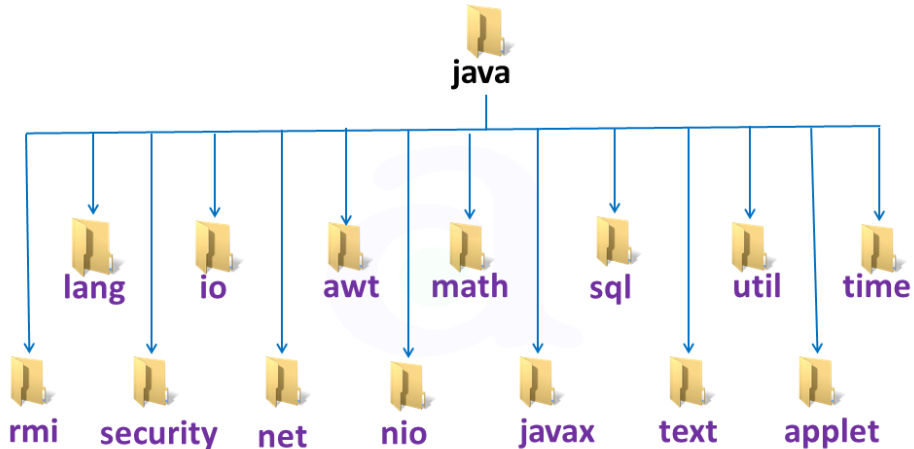













```
1 import java.sql.Date;  
2  
3 Data d1 = new Date() // java.sql.Date  
4 java.util.Date d2 = new java.util.Date();
```



```
1 package oop.localmods;  
2  
3 import java.awt.Point;  
4  
5 public class Test {  
6     public static void main(String[] args) {  
7         Point p = allocatePoint(2, 3);  
8     }  
9  
10    public static Point allocatePoint(int x, int y) {  
11        return new Point(x, y);  
12    }  
13 }
```



- 1 Using Objects
- 2 Designing Objects
- 3 Packages
- 4 References**

-  ALLEN B. DOWNEY, CHRIS MAYFIELD, ***THINK JAVA***, (2016).
-  GRAHAM MITCHELL, ***LEARN JAVA THE HARD WAY***, 2ND EDITION, (2016).
-  CAY S. HORSTMANN, ***BIG JAVA - EARLY OBJECTS***, 7E-WILEY, (2019).
-  JAMES GOSLING, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, ***THE JAVA LANGUAGE SPECIFICATION - JAVA SE 8 EDITION***, (2015).
-  MARTIN FOWLER, ***UML DISTILLED - A BRIEF GUIDE TO THE STANDARD OBJECT MODELING LANGUAGE***, (2004).
-  RICHARD WARBURTON, ***OBJECT-ORIENTED VS. FUNCTIONAL PROGRAMMING - BRIDGING THE DIVIDE BETWEEN OPPOSING PARADIGMS***, (2016).
-  NAFTALIN, MAURICE WADLER, PHILIP ***JAVA GENERICS AND COLLECTIONS***, O'REILLY MEDIA, (2009).
-  ERIC FREEMAN, ELISABETH ROBSON ***HEAD FIRST DESIGN PATTERNS - BUILDING EXTENSIBLE AND MAINTAINABLE OBJECT***, ORIENTED SOFTWARE-O'REILLY MEDIA, (2020).
-  ALEXANDER SHVETS ***DIVE INTO DESIGN PATTERNS***, (2019).

THANK YOU!