Java Programming Basics (I)

Desenvolvimento de Software e Sistemas Móveis (DSSMV)

Licenciatura em Engenharia de Telecomunicações e Informática

LETI/ISEP

2025/26

Paulo Baltarejo Sousa

pbs@isep.ipp.pt

Instituto Superior de Engenharia do Porto

P.PORTO

Disclaimer

Material and Slides

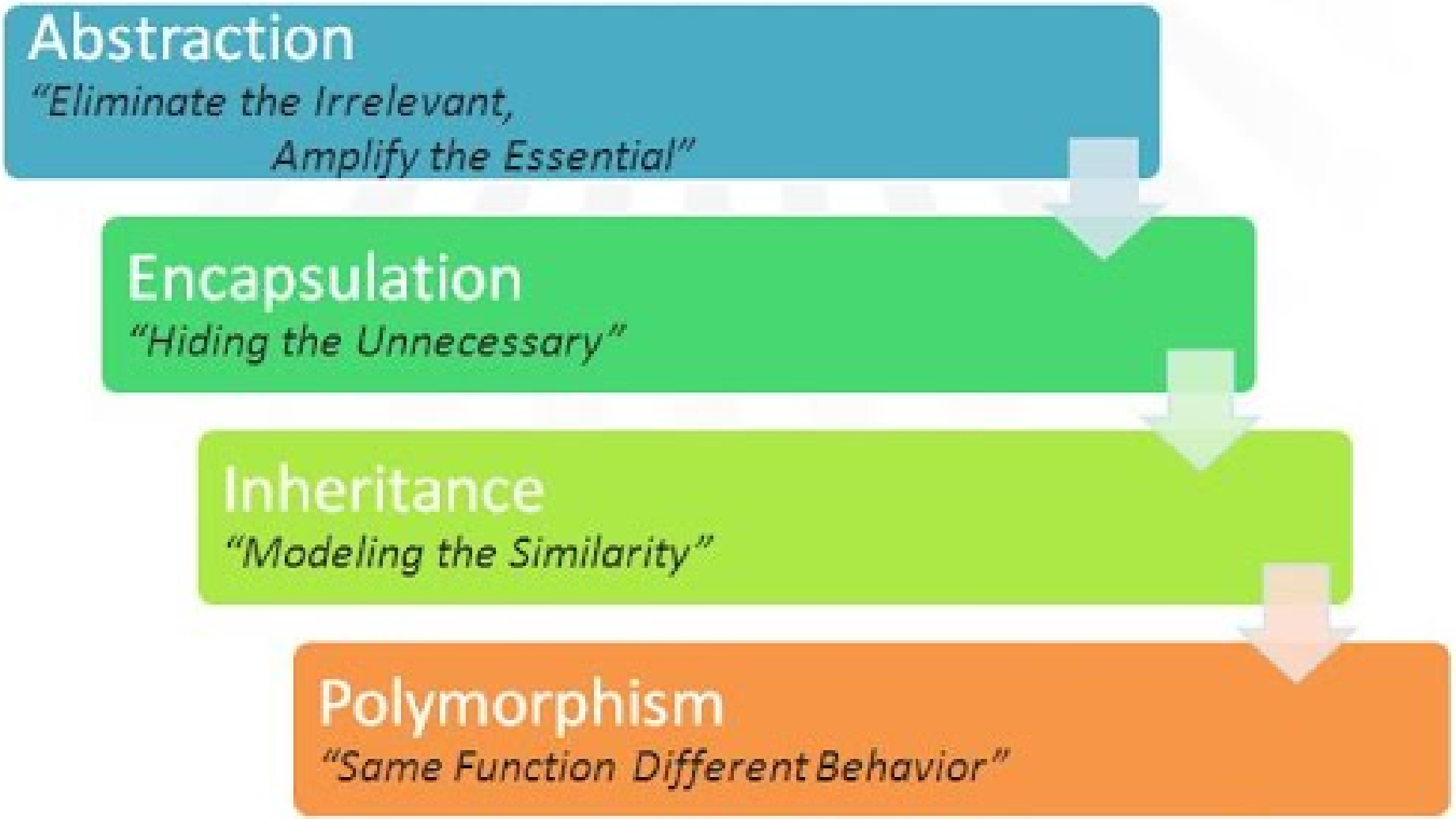
Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

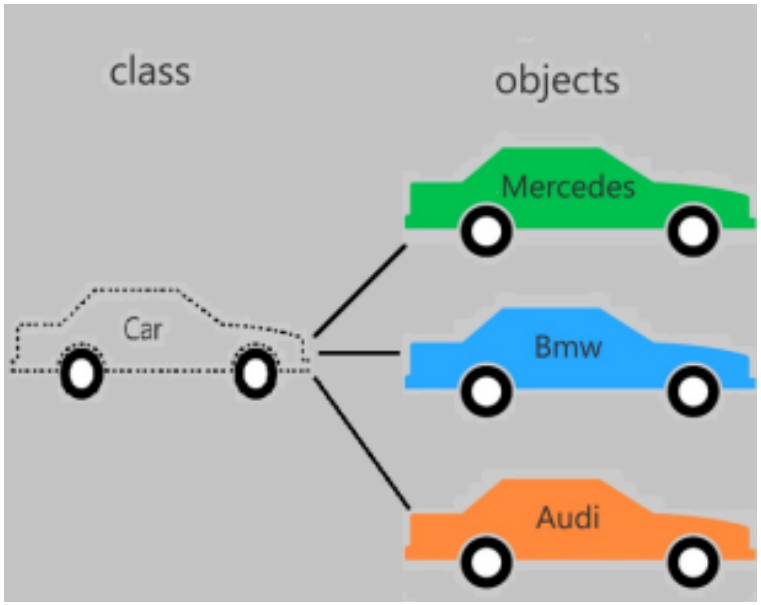
Programming Concepts

The OO Paradigm: Main characteristics



The OO Paradigm: Abstractions

- An abstraction is a simplified description of a system which captures the essential elements of that system while suppressing all other elements.
  - Abstractions are mostly classes.
- Programs are organized as collections of cooperative, dynamic **objects**, each of which is an instance of some **class**.



The OO Paradigm: What Is a Class?

- A **class is a blueprint or prototype** from which objects are created.
- **It is an abstraction of an object**, like `Bicycle`.
  - The fields `cadence`, `speed`, and `gear` represent the object's state,
  - The methods (`changeCadence`, `changeGear`, `speedUp` etc.) define its interaction with the outside world.

```
class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;
    void changeCadence(int newValue) {
        cadence = newValue;
    }
    void changeGear(int newValue) {
        gear = newValue;
    }
    void speedUp(int increment) {
        speed = speed + increment;
    }
    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}
```

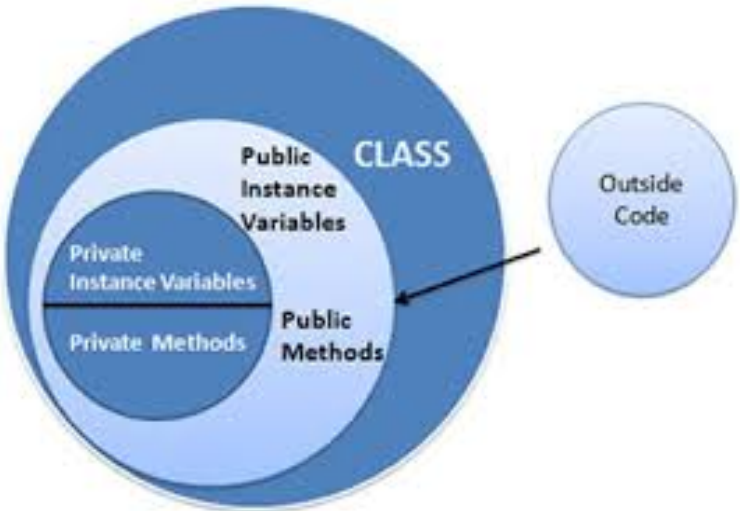
The OO Paradigm: What Is an Object?

- **It is an instantiation of a class**.
- It stores its state in **fields** (variables) and exposes its **behavior** through methods.
- **Methods** operate on an object's internal state and serve as the primary mechanism for object-to-object communication.
  - Hiding internal state and requiring all interaction to be performed through an object's methods is known as data **encapsulation**

```
class BicycleDemo {
   public static void main(String[] args)
         {
      // Create two different Bicycle
            objects
      Bicycle bike1 = new Bicycle();
      Bicycle bike2 = new Bicycle();
      // Invoke methods on those objects
      bike1.changeCadence(50);
      bike1.speedUp(10);
      bike1.changeGear(2);
      bike2.changeCadence(50);
      bike2.speedUp(10);
      bike2.changeGear(2);
   }
}
```

The OO Paradigm: Encapsulation

- We **encapsulate the details** of a class **inside its private part** so that it is impossible for any of the class's clients to know about or depend upon these details.
- The **ability to change the representation of an abstraction (data structures, algorithms) without disturbing any of its clients** is the essential benefit of encapsulation.



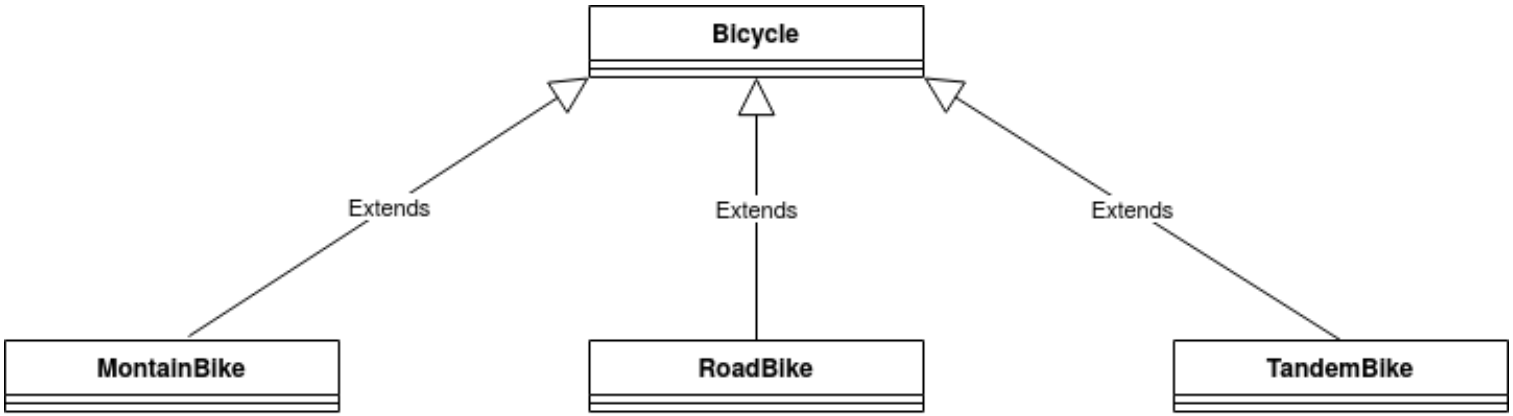

```
class Bicycle {
   private int cadence = 0;
   private int speed = 0;
   private int gear = 1;
   public void changeCadence(int newValue) { ... }
   public void changeGear(int newValue) { ... }
   public void speedUp(int increment) { ... }
   public void applyBrakes(int decrement) { ... }
}
```

The OO Paradigm: What Is Inheritance? (I)

- **Inheritance** provides a powerful and natural mechanism for organizing and structuring your software.
- Different kinds of objects often have a certain amount in common with each other.
  - Mountain bikes, road bikes, and tandem bikes, for example, all share the characteristics of bicycles (current speed, current pedal cadence, current gear).
    - Yet each also defines additional features that make them different: tandem bicycles have two seats and two sets of handlebars; road bikes have drop handlebars; some mountain bikes have an additional chain ring, giving them a lower gear ratio.

The OO Paradigm: What Is Inheritance? (II)

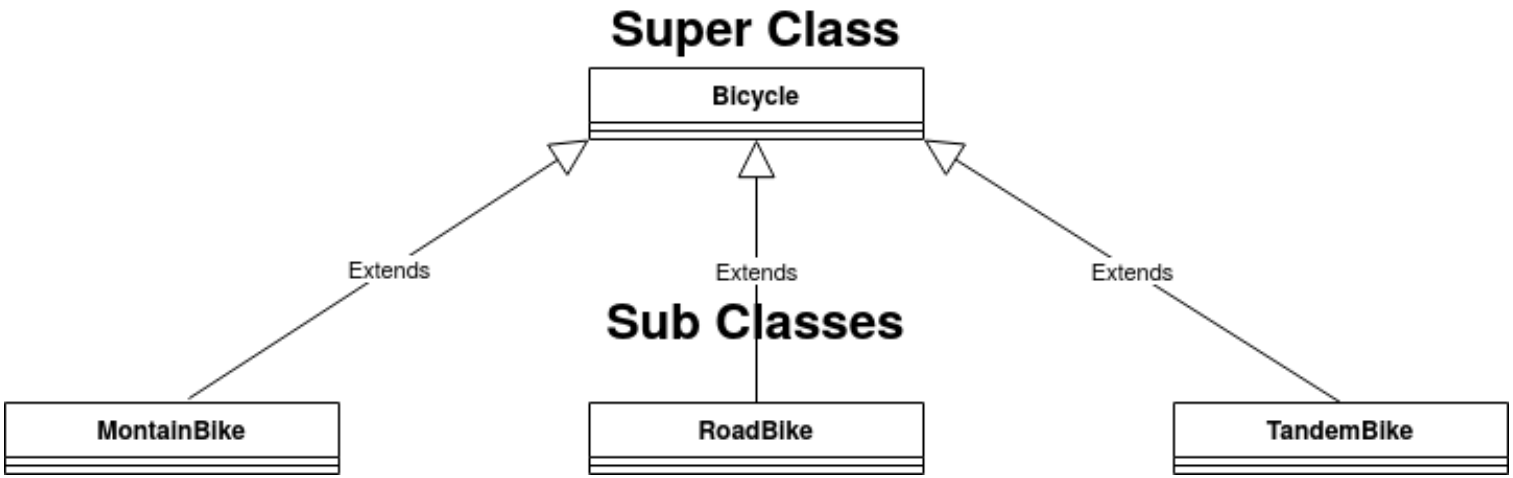
```
class MountainBike extends Bicycle {

    // new fields and methods defining
    // a mountain bike would go here
}
```



Text

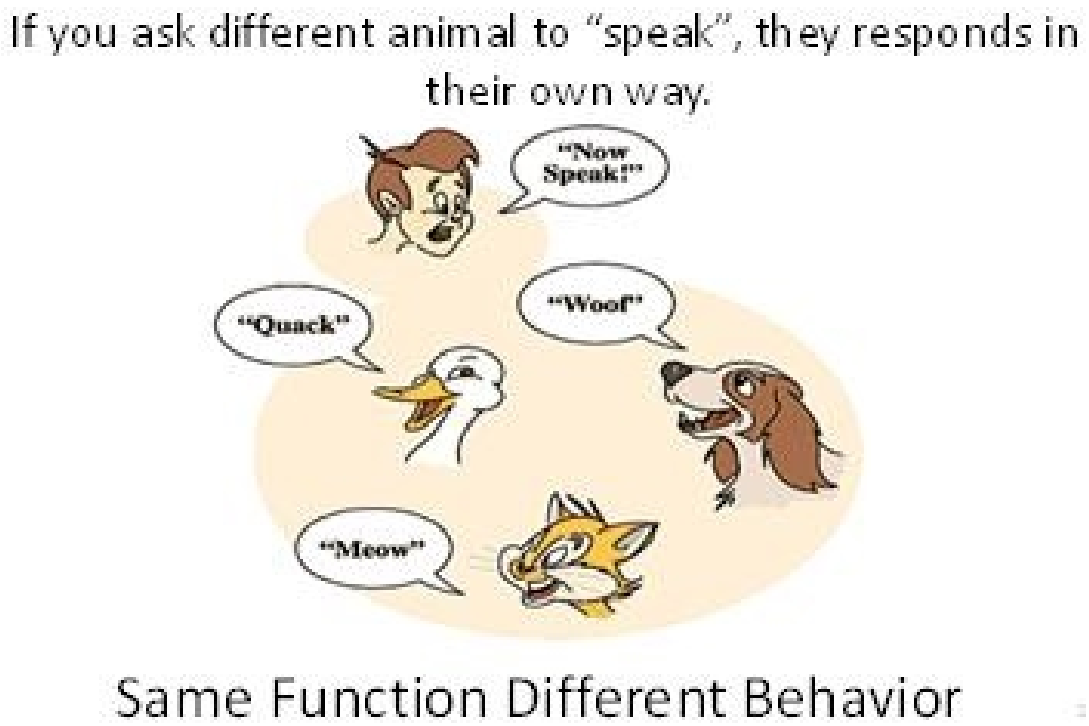
The OO Paradigm: What Is Inheritance? (III)

- OO Paradigm allows classes to inherit commonly used state and behavior from other classes.
- Each class is allowed to have one direct superclass, and each superclass has the potential for an unlimited number of subclasses.



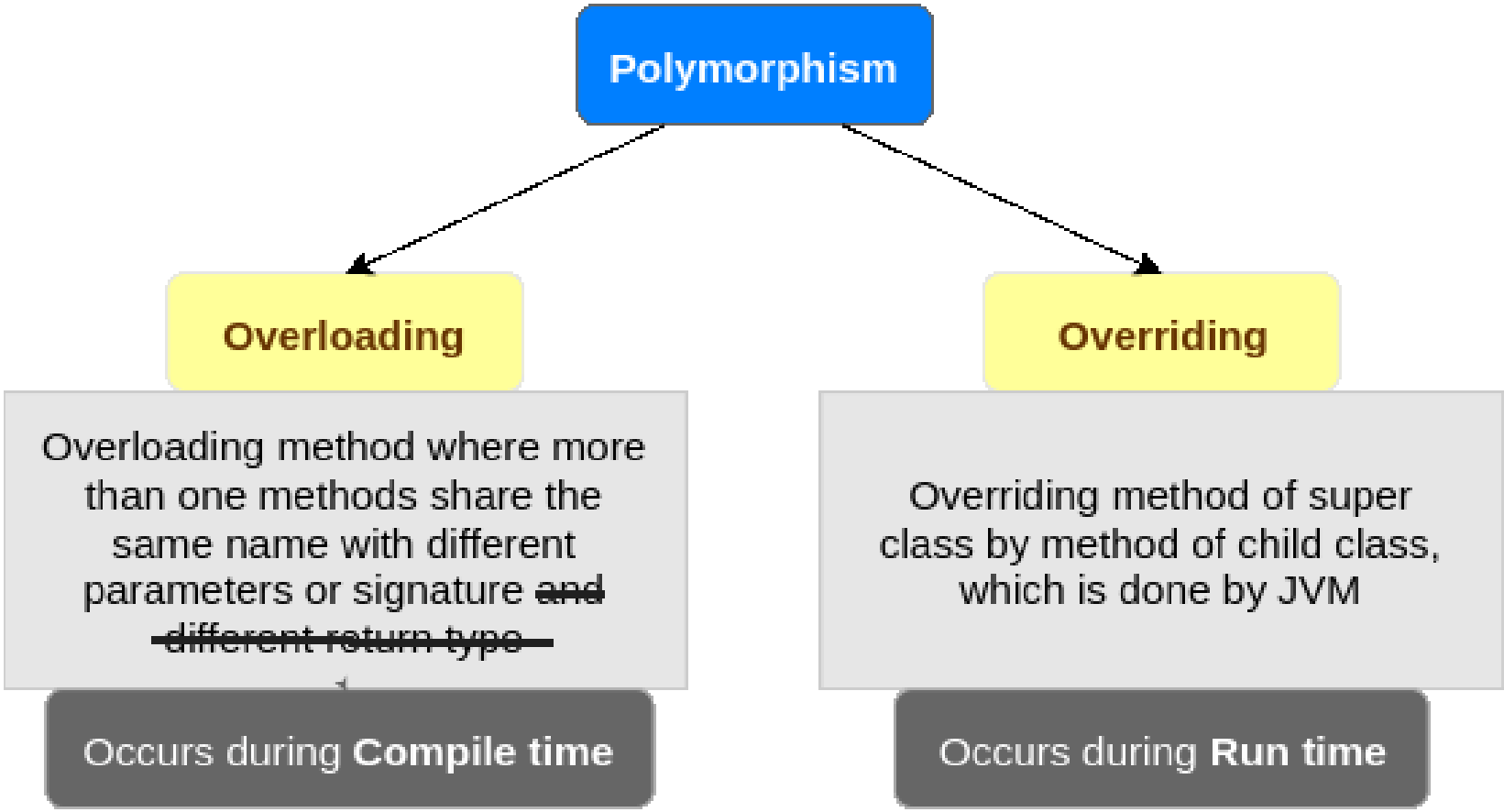
The OO Paradigm:Polymorphism (I)

- The word “Polymorphism” comes from two Greek words, “many” and “form”.
  - We can illustrate the idea of polymorphism easily using the scenario where different animals are asked to “speak”.



The OO Paradigm:Polymorphism (II)

- **Overloading** is static polymorphism.
- **Overriding** is dynamic polymorphism.



The OO Paradigm:Polymorphism: Overloading

- This allows us to have more than one method having the same name, if the parameters of methods are different in number, sequence and data types of parameters.

```
class SimpleCalculator
{
   int add(int a, int b)
   {
       return a+b;
   }
   int add(int a, int b, int c)
   {
       return a+b+c;
   }
}
```

The OO Paradigm:Polymorphism: Overriding

```
public class ABC{
   public void myMethod(){
      System.out.println("Overridden Method: ABC");
   }
}
```

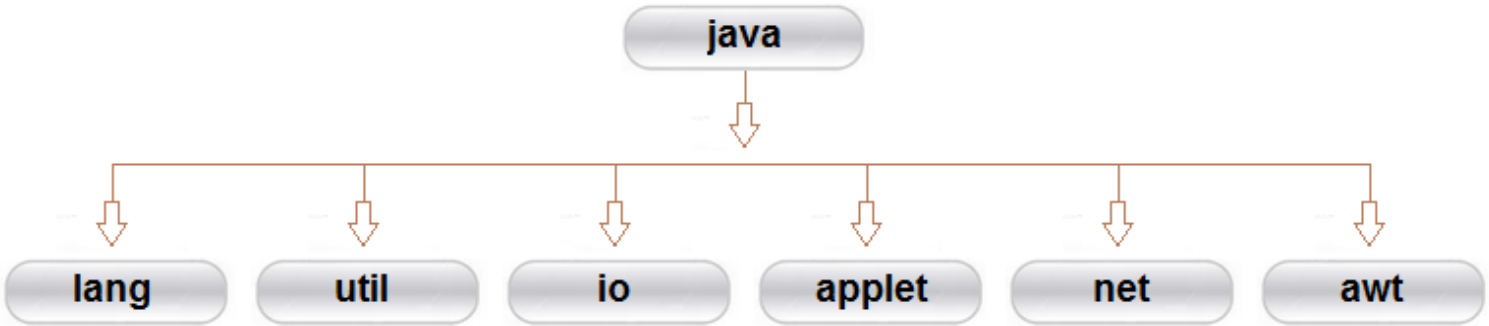
```
public class XYZ extends ABC{
   public void myMethod(){
      System.out.println("Overriding Method: XYZ");
   }
}
```

```
public class Main{
   public static void main(String args[]){
      ABC obj1 = new ABC();
      obj1.myMethod(); // This would call the myMethod() of parent class ABC
      XYZ obj2 = new XYZ();
      obj2.myMethod(); // This would call the myMethod() of child class XYZ
      ABC obj3 = new XYZ();
      obj3.myMethod(); // This would call the myMethod() of child class XYZ
   }
}
```

Java Basics

What Is a Package?

- A package is a **namespace for organizing classes and interfaces** in a logical manner.
  - A package is a collection of classes with a related purpose.
  - Java classes are grouped into packages. Use the `import` statement to use classes that are declared in other packages.
- Conceptually you can think of **packages as being similar to different folders on your computer**.
- Java library consists of several hundred packages.



Creating a Package

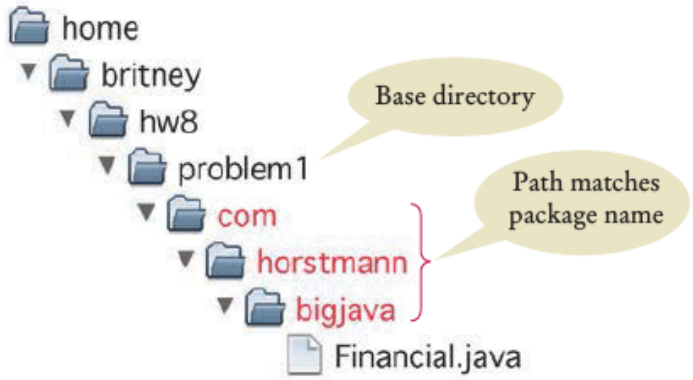
- To create a package, you choose a name for the package and put a `package` statement with that name at the top of every source file that contains the types that you want to include in the package.

Syntax `package packageName;`

```
package com.horstmann.bigjava;
```

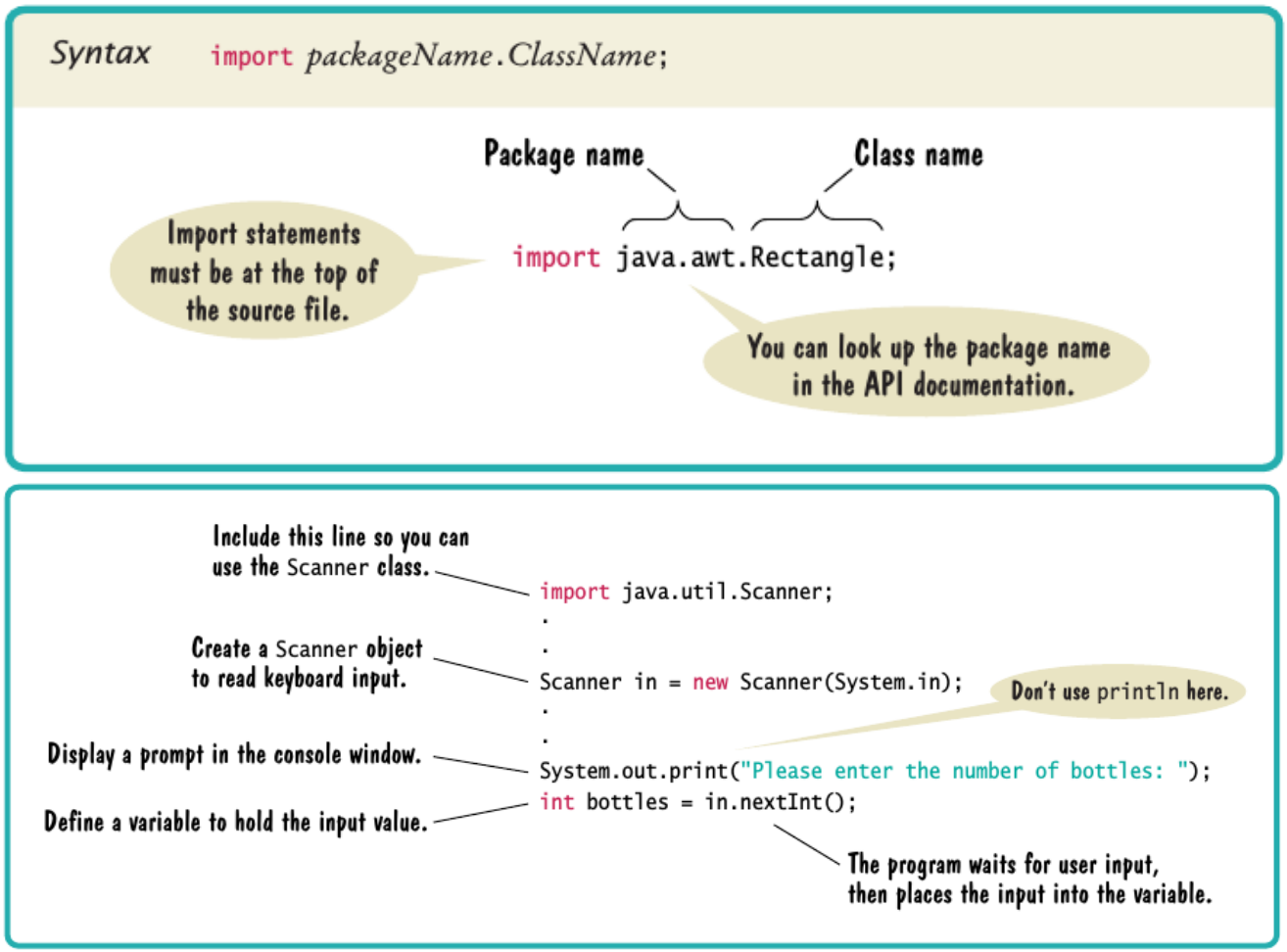
The classes in this file belong to this package.

A good choice for a package name is a domain name in reverse.



Using a Package

- Use the `import` statement to use classes that are declared in other packages.



Class

- A Java application is composed by a set of classes.

Syntax

```
accessSpecifier class ClassName
{
   instance variables
   constructors
   methods
}
```

```
public class Counter
{
   private int value;

   public Counter(int initialValue) { value = initialValue; }

   public void click() { value = value + 1; }
   public int getValue() { return value; }
}
```

Public interface

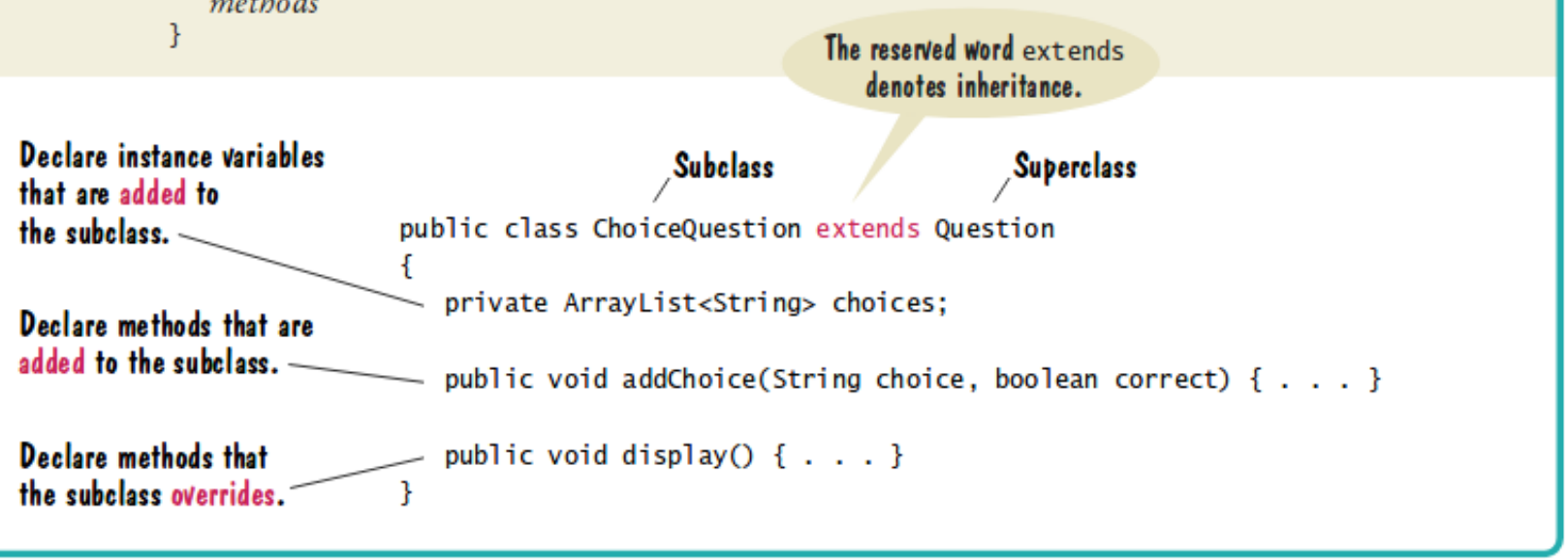
Private implementation

Subclass

- A class that inherits variables and methods from a superclass but may also add instance variables, add methods, or redefine methods.

Syntax

```
public class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```



What Is an Interface? (I)

- An **interface** is a contract between a class and the outside world.
  - When a class implements an interface, it promises to provide the behavior published by that interface.
- An **interface is a group of related methods with empty bodies** (abstract).

```
interface Bicycle {
   void changeCadence(int newValue);
   void changeGear(int newValue);
   void speedUp(int increment);
   void applyBrakes(int decrement);
}
```

- Implementing an interface allows a class to become more formal about the behavior it promises to provide.

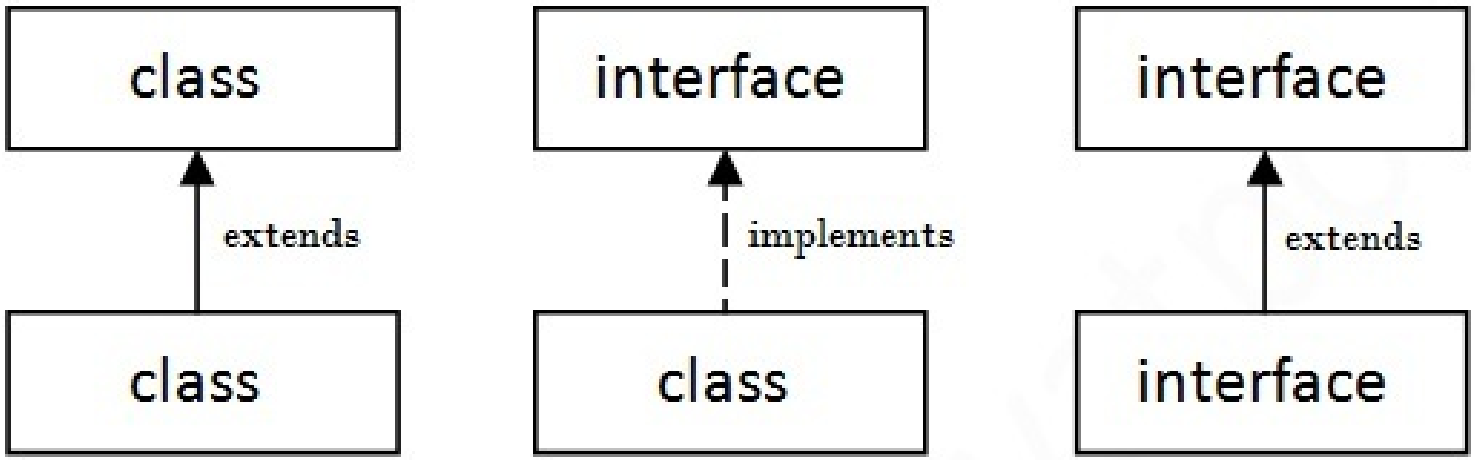
What Is an Interface? (II)

- If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

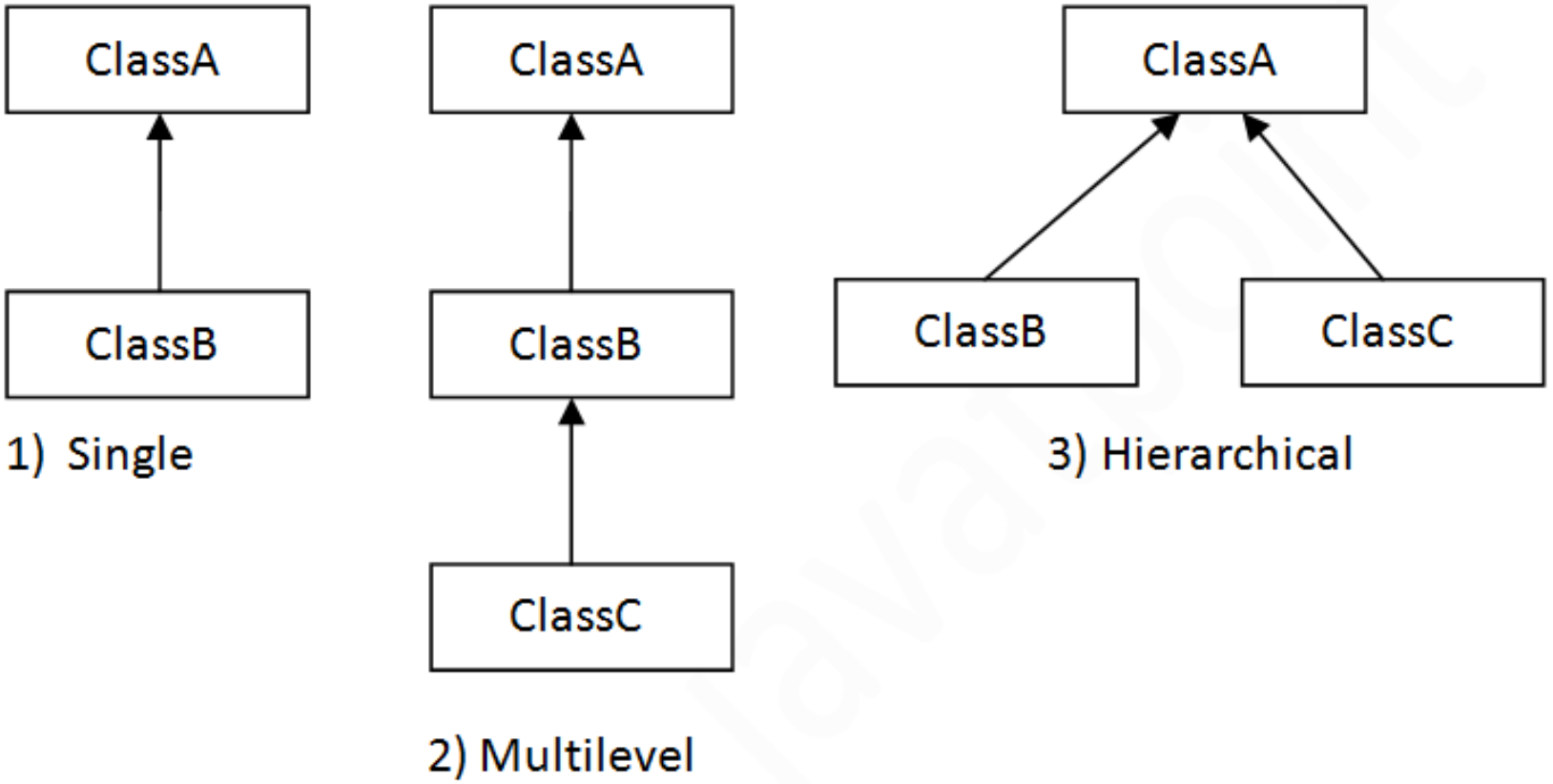
```
class MountainBike implements Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }
    void changeGear(int newValue) {
        gear = newValue;
    }
    void speedUp(int increment) {
        speed = speed + increment;
    }
    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }
}
```

Classes and Interfaces

- A class **extends** another class, an interface **extends** another interface but a class **implements** an interface.



Types of inheritance (I)



- There can be three types of inheritance: single, multilevel and hierarchical.

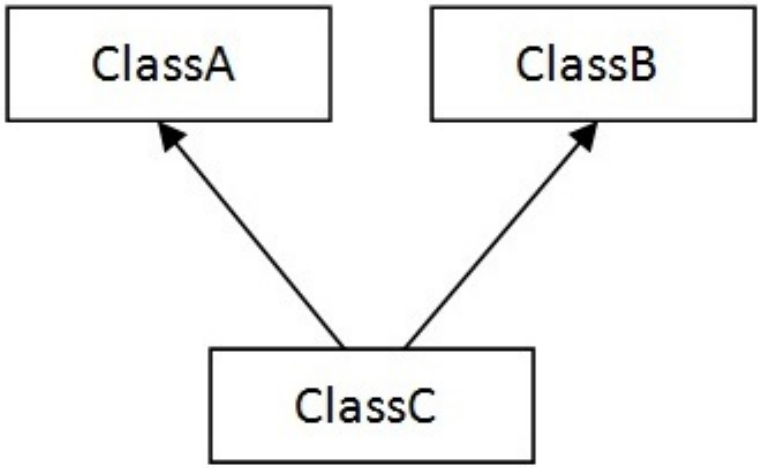
Types of inheritance (II)

```
class Animal{
    ...
}
class Dog extends Animal{
    ...
}
```

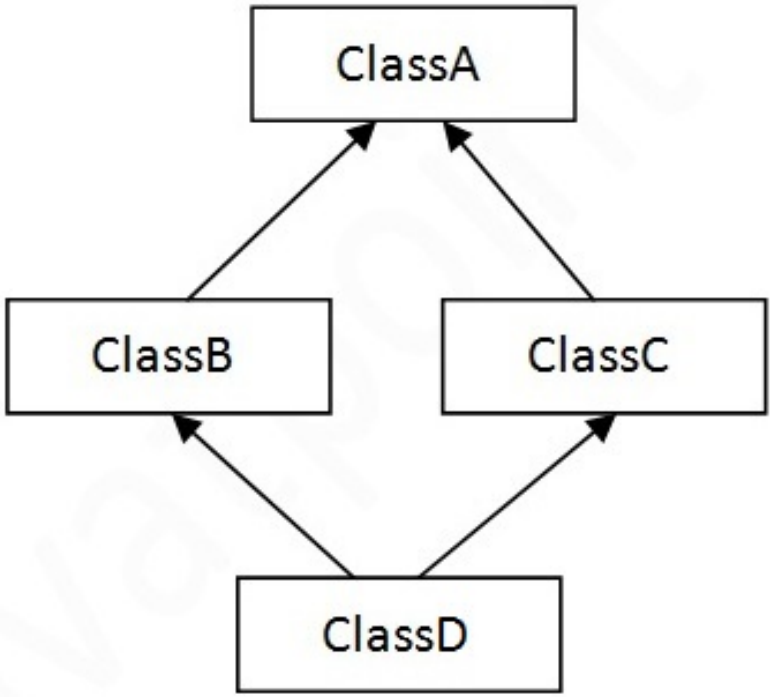
```
class Animal{
    ...
}
class Dog extends Animal{
    ...
}
class BabyDog extends Dog{
    ...
}
```

```
class Animal{
    ...
}
class Dog extends Animal{
    ...
}
class Cat extends Animal{
    ...
}
```

Types of inheritance (III)



4) Multiple



5) Hybrid

- Multiple and hybrid inheritance is supported through **interface** only

Types of inheritance (IV)

```
interface A {
  public void methodA();
}
interface B {
  public void methodB();
}
class MultipleInheritance implements A,B{
  public void methodA() {
    System.out.println("Calling methodA");
  }
  public void methodB() {
    System.out.println("Calling methodB");
  }
}
```

```
interface A {
  public void methodA();
}
interface B extends A {
  public void methodB();
}
interface C extends A {
  public void methodC();
}
class HybridInheritance implements B, C {
  public void methodA() {
    System.out.println("Calling methodA");
  }
  public void methodB() {
    System.out.println("Calling methodB");
  }
  public void methodC() {
    System.out.println("Calling methodC");
  }
}
```

A Closer Look at Methods

Overloading Methods (I)

- In Java it is possible **to define two or more methods within the same class that share the same name**, as long as their parameter declarations are different.
  - When this is the case, the **methods are said to be overloaded**, and the process is referred to as method overloading.
- Method overloading is one of the ways that Java supports **polymorphism**.
- When an overloaded method is invoked, Java uses the **type** and/or **number of arguments** as its guide to determine which version of the overloaded method to actually call.
  - Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different **return types**, the return type alone **is insufficient to distinguish two versions of a method**.

Overloading Methods (II)

```
class OverloadDemo {
   void test() {
      System.out.println("No parameters");
   }
// Overload test for one integer parameter.
   void test(int a) {
      System.out.println("a: " + a);
   }
// Overload test for two integer parameters.
   void test(int a, int b) {
      System.out.println("a and b: " + a + " " + b);
   }
// overload test for a double parameter
   double test(double a) {
      System.out.println("double a: " + a);
      return a*a;
   }
}
```

Overloading Constructors

```
class Box {
   double width;
   double height;
   double depth;
// constructor used when all dimensions specified
   Box(double w, double h, double d) {
      this.width = w;
      this.height = h;
      this.depth = d;
   }
// constructor used when no dimensions specified
// use -1 to indicate an uninitialized box
   Box() {
      this.width = -1;
      this.height = -1;
      this.depth = -1;
   }
// constructor used when cube is created
   Box(double len) {
      this.width = this.height = this.depth = len;
   }
}
```

Using Objects as Parameters

- Methods and constructors

```
class Box {
   double width;
   double height;
   double depth;
   // This is constructor for Box that receive an object.
   Box(Box obj) {
      width = obj.width;
      height = obj.height;
      depth = obj.depth;
   }
   // return true if o is equal to the invoking object
   boolean equals(Box obj) {
      if( this. width == obj.width && this.height == obj.height && this.depth == obj.
            depth)
         return true;
      else
         return false;
   }
}
```

A Closer Look at Argument Passing

- In general, there are two ways that a computer language can pass an argument to a method:
  - The first way is **call-by-value**.
    - This approach copies the value of an argument into the formal parameter of the method.
    - Therefore, **changes made to the parameter of the method have no effect on the argument**.
  - The second way an argument can be passed is **call-by-reference**.
    - In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the method, this reference is used to access the actual argument specified in the call.
    - This means that **changes made to the parameter will affect the argument used to call the method**.
- In Java
  - When you pass a **primitive type** to a method, **it is passed by value**.
  - When you pass an **object** to a method, **it is passed by reference**.

A Closer Look at Returning Types (I)

- **A method can return any type of data**, including class and primitive types as well as `void`.

```
public class Box {
   private double width;
   private double height;
   private double depth;
   public Box(double width, double height, double depth) {
      this.width = width;
      this.height = height;
      this.depth = depth;
   }
   public double getWidth() {
      return width;
   }
   public void setWidth(double width) {
      this.width = width;
   }
   public double getVolume() {
      return (width * height * depth);
   }
   public Box getBoxObject() {
      Box box = new Box(10,10, 10);
      return box;
   }
}
```

A Closer Look at Returning Types (II)

- Returning a **value**

```
public class Box {
...
   public double getWidth() {
      return width;
   }
   ...
}
```

- Returns an object **reference**
  - All objects are dynamically allocated in the **heap** using `new` operator.
    - The object will continue to exist as long as there is a reference to it somewhere in your program.

```
public class Box {
...
   public Box getBoxObject() {
      Box box = new Box(10,10, 10);
      return box;
   }
}
```

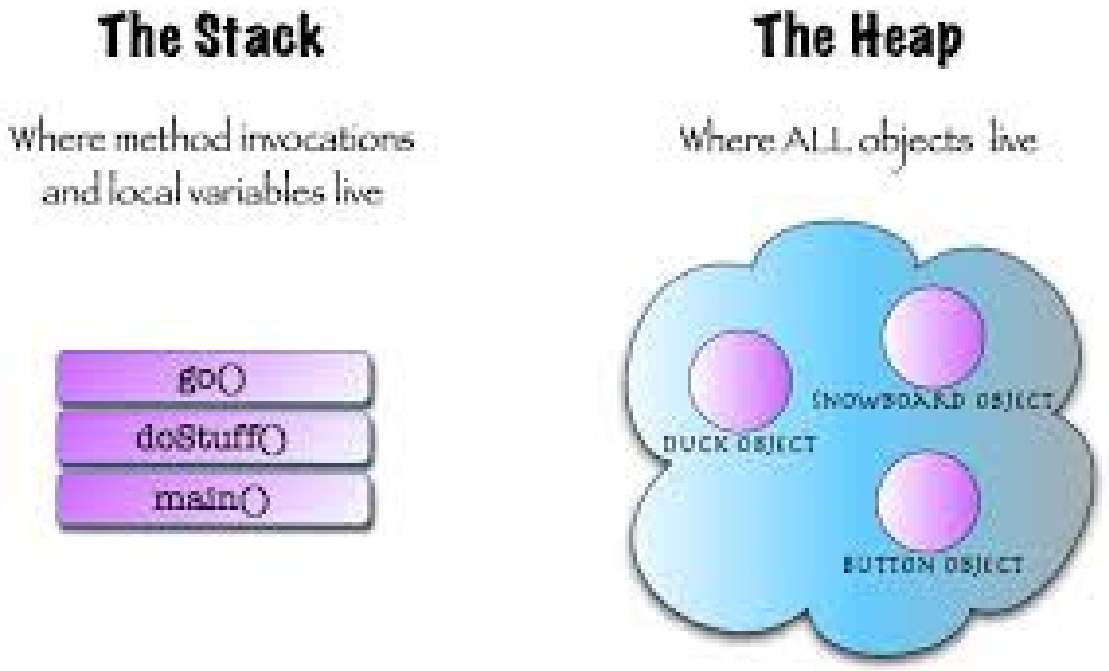
Memory Utilization in Java

How CPU memory works?

- Every time CPU executes the given instructions, it stores the results in registers.
  - Registers are the closest memory to CPU and hence the fastest.
- The next series in the CPU memory is RAM, which is accessible to CPU by the memory bus.
  - A CPU can access the physical memory, RAM.
- The orchestration of memory is controlled by the underline Operating System (OS), which maps the physical memory to each process's memory.

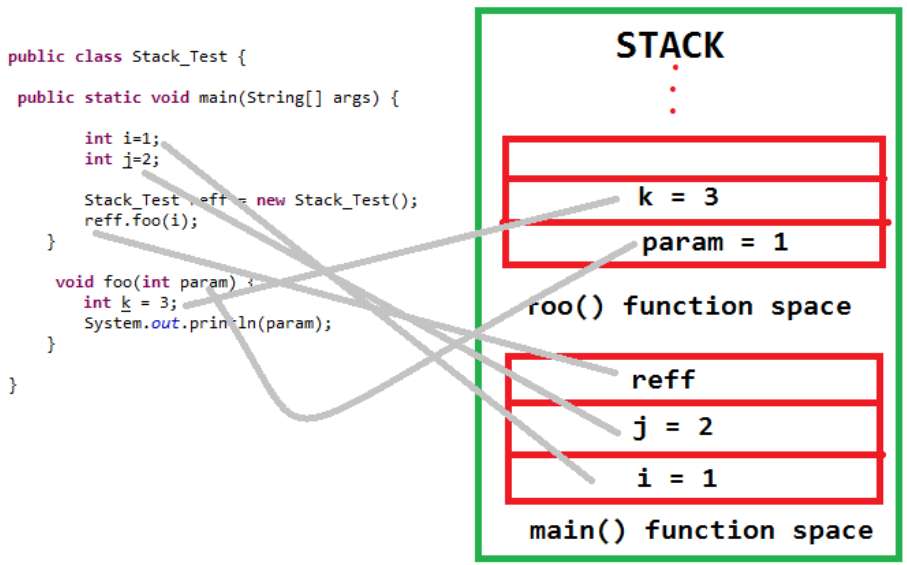
Java Memory Model

- It specifies how the Java Virtual Machine (JVM) works with the computer's memory (RAM).
- It specifies the memory management rules.
- The memory used internally in the JVM is divided into **stack** and **heap**.



What is Java Stack?

- **Stack is the home of local variables and method invocations**.
- Stack manages order as **Last-In-First-Out** (LIFO).
  - The top of the stack contains the currently running method.
  - Your code calls a method, then its **stack frame** gets created and it gets put onto the top of the call stack.
  - This stack frame maintains the state of the method.
    - **It also keeps track of the line of code currently getting executed and the values of all the local variables**.



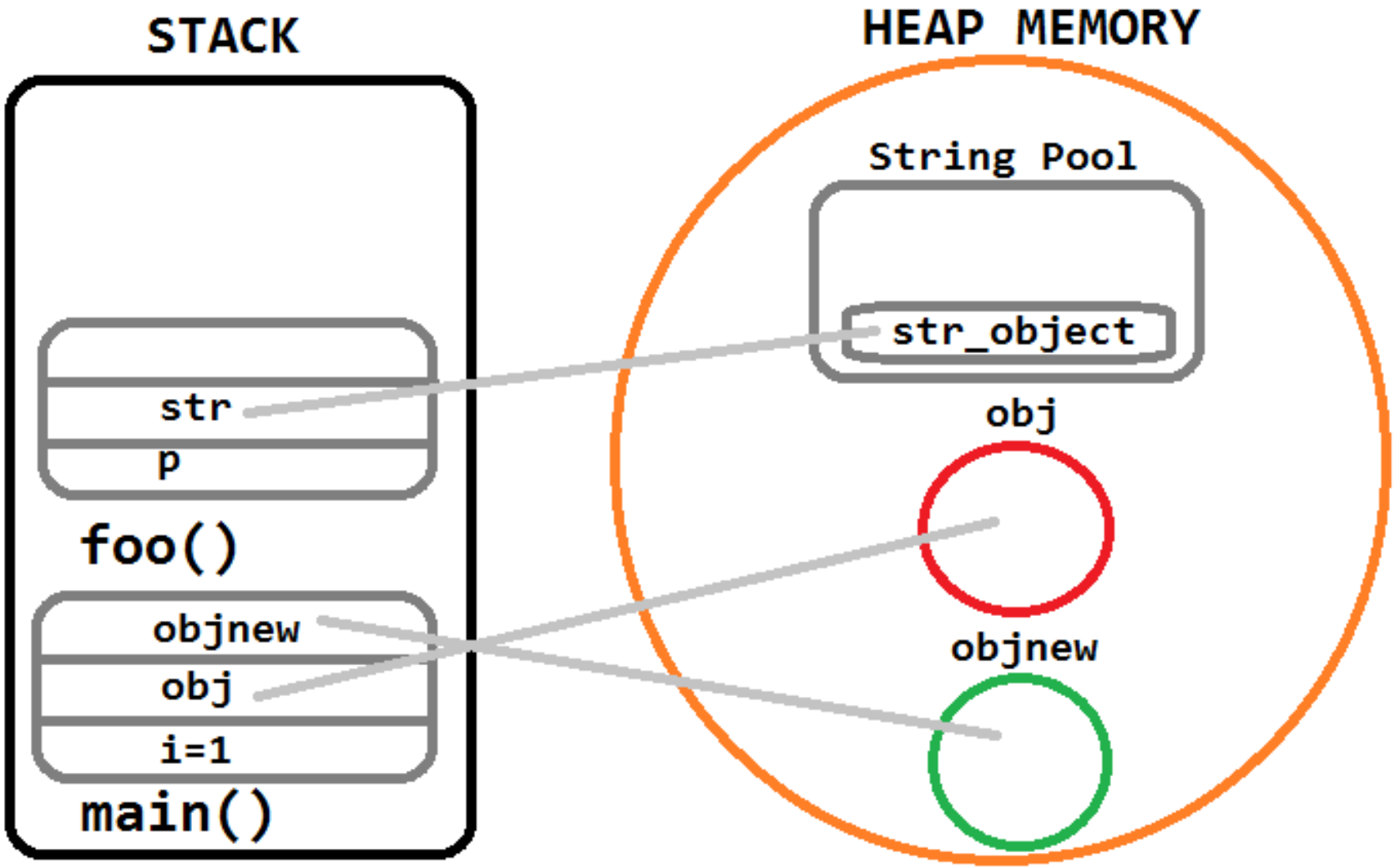
What is Java Heap?

- The **heap is used for storing all the objects created by using the operator `new`**.
- As long as you have an active reference to this newly created object, it uses memory space in heap.
- Once you **remove this reference, this object becomes an orphan** and available for removal so that JVM can claim this allocated memory and provide it to other objects.
  - In order to clean this dead reference, Java has provided a tool called **garbage collector**.
  - **This garbage collector then collects all the orphan objects and frees up the memory** that JVM can reuse.

Stack and Heap (I)

```
public class Heap_Stack {
      public static void main(String[] args) {
         // primitive datatype created inside main() method space in stack memory
         int i=1;
         // Object created in heap memory and its refference obj in stack memory
         Object obj = new Object();
         // Heap_Stack Object created in heap memory and its refference objnew in
               stack memory
         Heap_Stack objnew = new Heap_Stack();
         // New space for foo() method created in the top of the stack memory
         objnew.foo(obj);
      }
      private void foo(Object p) {
        // String for p.toString() is created in String Pool and refference str
             created in stack memory
         String str = p.toString();
         System.out.println(str);
      }
}
```

Stack and Heap (II)



Bibliography

Resources

- "Big Java: Early Objects", 6th Edition by Cay S. Horstmann
- "Java™:The Complete Reference", 7th Edition,Herbert Schildt
- "Java™Programming", 7th Edition, Joyce Farrell
- `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html`
- `http://beginnersbook.com/java-tutorial-for-beginners-with-examples/`
- `https://www.leepoint.net/index.html`
- `https://junit.org/junit5/docs/current/user-guide/`