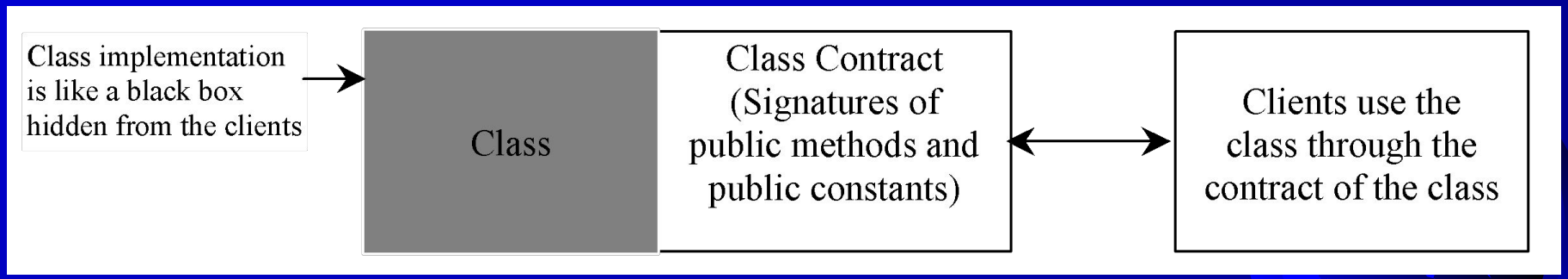
Class & Objects (Continues..)

Principles of OOP

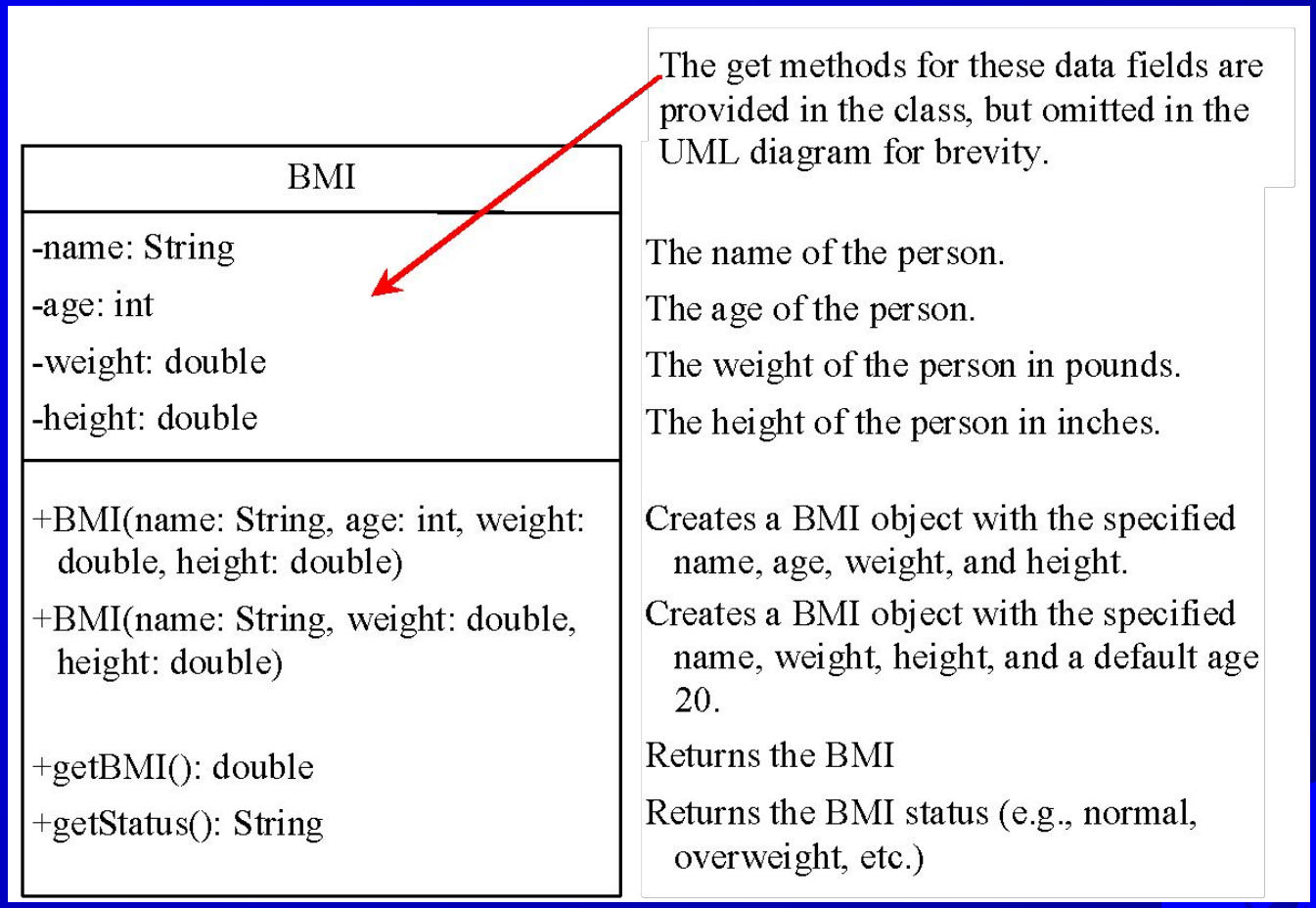
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the user of the class. The creator of the class provides a description of the class and let the user know how the class can be used. The user of the class does not need to know how the class is implemented. The detail of implementation is encapsulated and hidden from the user.



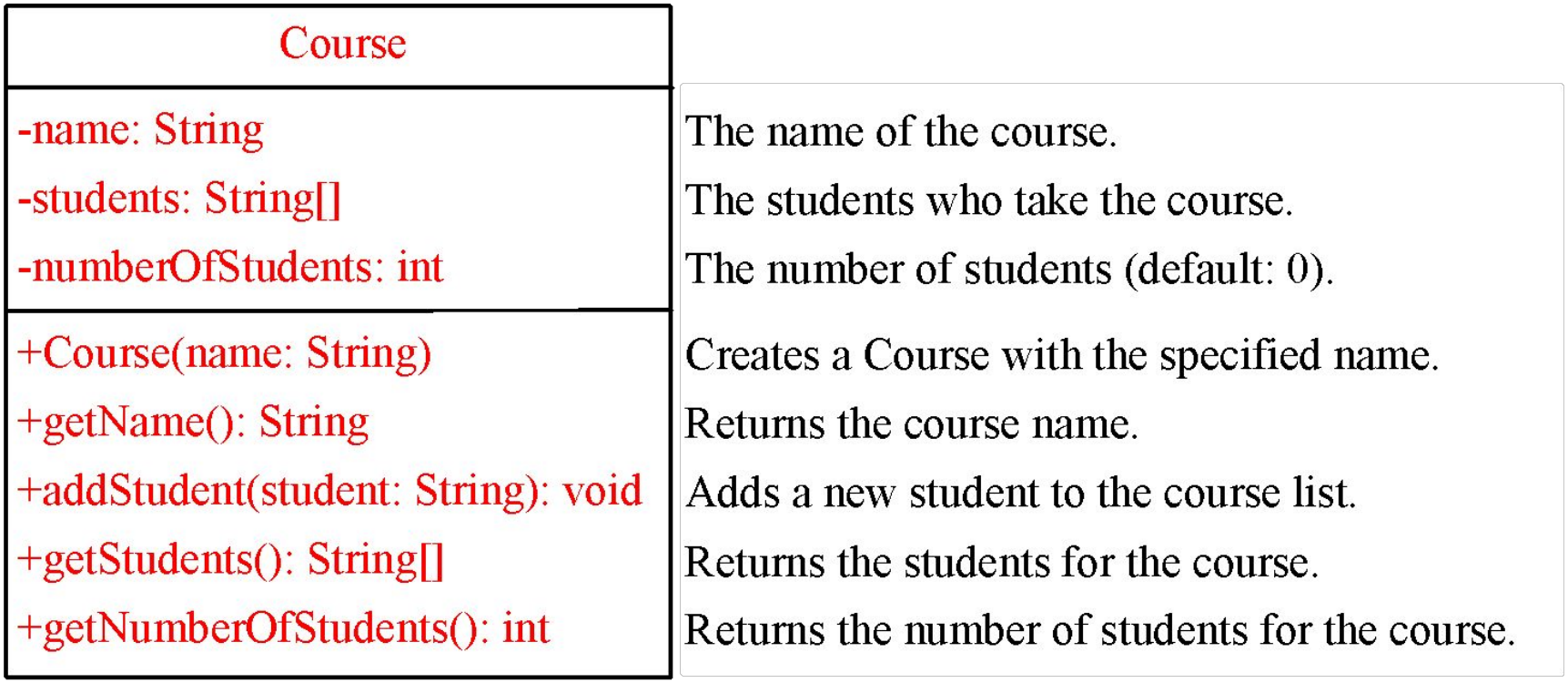
The BMI Class



BMI

UseBMIClass

Example: The Course Class



Course TestCource Run

Encapsulation

- A process of hiding all the internal details of an object from the outside real world.
- Like enclosing into the capsule.
- Restricts client from seeing implementation detail.

Encapsulation (Example)

Examples:

- if someone wants to know my name then he cannot directly access my brain cells to get to know what is my name. Instead that person will either ask my name.

- If a driver wants to speed up a vehicle then he doesn't start to put more gas/oil inside the engine. He uses the interface (accelerator pedal, gear, etc) for that purpose.

Encapsulation in Java

The variables of a class might need to be hidden from other classes, and can be accessed only through the methods of their current class, it is also known as *data hiding*.

To achieve encapsulation (data hiding) in Java

- Declare the variables of a class as *private*.
- Provide public *setter methods* (also known as *accessors*) and *getter methods* (also known as *mutators*) to write and read the variables values.

Encapsulation in Java (Example)

```
public class EncapsulationTest{

 private String name;
 private int age;

   public int getAge(){
      return age;
   }

   public String getName(){
      return name;
   }

   public void setAge( int newAge){
      age = newAge;
   }

   public void setName(String newName){
      name = newName;
   }
}
```

Encapsulation in Java (Example)

```
public class RunEncapsulation{

   public static void main(String args[]){
      EncapsulationTest encap = new EncapsulationTest();
      encap.setName("James");
      encap.setAge(20);

      System.out.print("Name : " + encap.getName() + " Age : "
                       + encap.getAge());
    }
}
```

```
Output:
Name : James Age : 20
```

Designing a Class

- (Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

Designing a Class, cont.

- (Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
- The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

Designing a Class, cont.

- Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.

Designing a Class, cont.

- Follow standard Java programming style and naming conventions.
- Choose informative names for classes, data fields, and methods.
- Always place the data declaration before the constructor, and place constructors before methods.
- Always provide a constructor and initialize variables to avoid programming errors.

The this Keyword
Reference the Hidden Data Fields

The this keyword is the name of a reference that refers to an object itself. One common use of the this keyword is reference a class's *hidden data fields*.

```
public class Foo {
  private int i = 5;
  private static double k = 0;

  void setI(int i) {
    this.i = i;
  }

  static void setK(double k) {
    Foo.k = k;
  }
}
```

```
Suppose that f1 and f2 are two objects of Foo.

Invoking f1.setI(10) is to execute
   this.i = 10, where this refers f1

Invoking f2.setI(45) is to execute
   this.i = 45, where this refers f2
```

Calling Overloaded Constructor

Another common use of the this keyword to enable a constructor to invoke another constructor of the same class.

```
public class Circle {
  private double radius;

  public Circle(double radius) {
    this.radius = radius;
  }

  public Circle() {
    this(1.0);
  }

  public double getArea() {
    return this.radius * this.radius * Math.PI;
  }
}
```

this must be explicitly used to reference the data field radius of the object being constructed

this is used to invoke another constructor

Every instance variable belongs to an instance represented by this, which is normally omitted