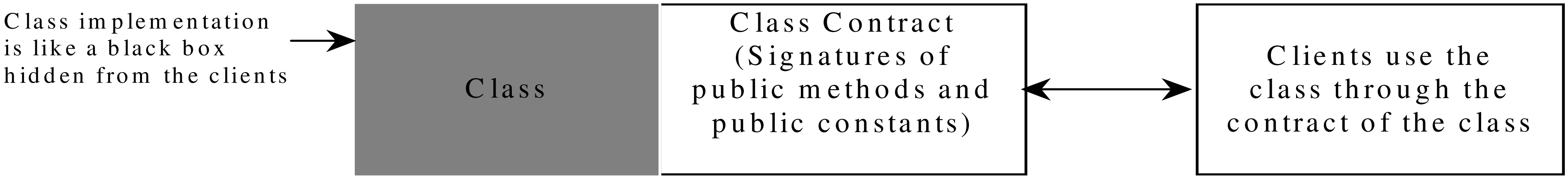
Chapter 10 - Thinking in Objects

Dr. ASEM KITANA
Dr. ABDALLAH KARAKRA

Class Abstraction and Encapsulation

Class abstraction means to separate class implementation from the use 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.


Class implementation is like a black box hidden from the clients
Class
Class Contract (Signatures of public methods and public constants)
Clients use the class through the contract of the class

Class Relationships

- Association
- Aggregation
- Composition
- Inheritance (**Later**)

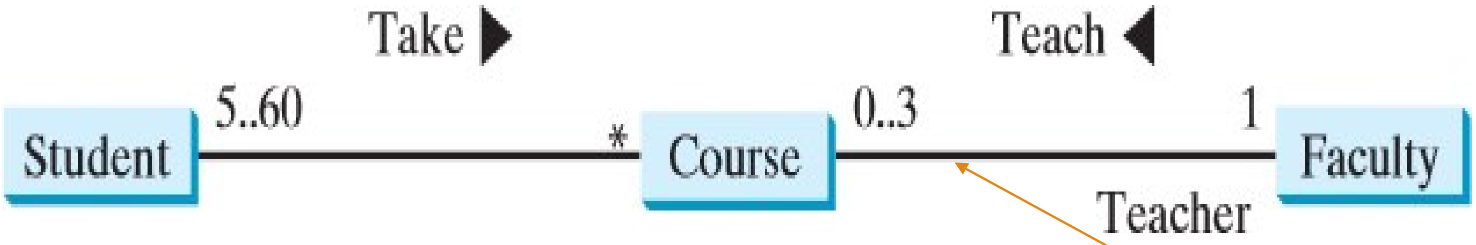
Association

Association is a general binary relationship that describes an activity between two classes

Examples:

- a student taking a course is an association between the Student class and the Course class
- faculty member teaching a course is an association between the Faculty class and the Course class

Association



a solid line between two classes

This UML diagram shows that

1. a student may take any number of courses
2. a faculty member may teach at most three courses
3. a course may have from five to sixty students
4. a course is taught by only one faculty member

Notes

- An association is illustrated by **a solid line between two classes** with an **optional label**

the labels are **Take** and **Teach**

- Each relationship may have an **optional small black triangle** that indicates **the direction of the relationship**

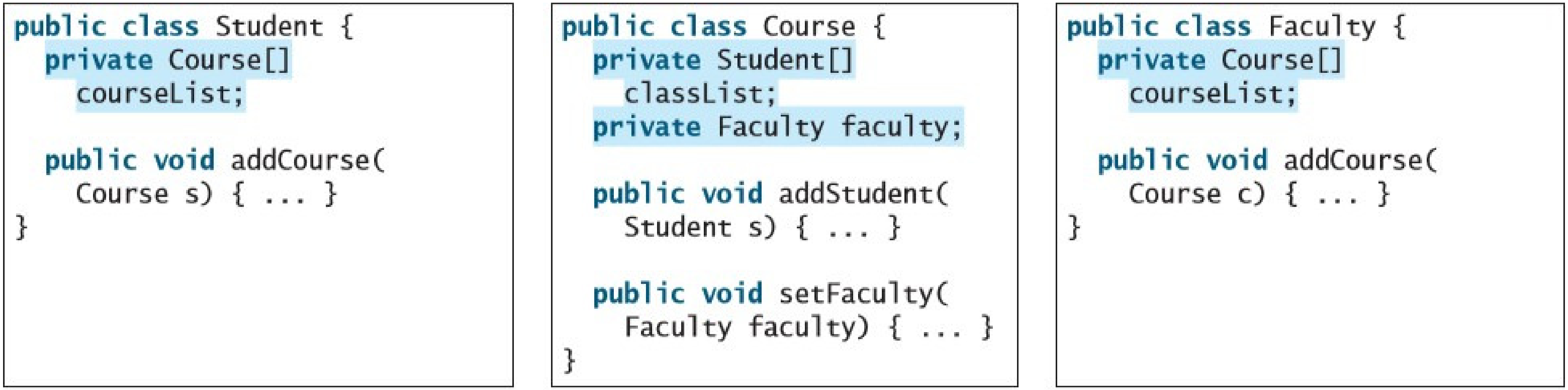
- Each class involved in the **relationship may have a role name** that describes the role it plays in the relationship. In previous Figure , **teacher is the role name for Faculty.**

- **Each class involved in an association may specify a multiplicity, which is placed at the side of**
the class to specify how many of the class's objects are involved in the relationship in UML

A multiplicity could be a number or an interval that specifies how many of the class's objects are involved in the relationship

The character * means an unlimited number of objects, and the interval m..n indicates that the number of objects is between m and n, inclusively.

Association in Javacode



```java
public class Student {
  private Course[]
    courseList;

  public void addCourse(
    Course s) { ... }
}
```

```java
public class Course {
  private Student[]
    classList;
  private Faculty faculty;

  public void addStudent(
    Student s) { ... }

  public void setFaculty(
    Faculty faculty) { ... }
}
```

```java
public class Faculty {
  private Course[]
    courseList;

  public void addCourse(
    Course c) { ... }
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

Association in Javacode

In Java code, you can implement associations by using data fields and methods

The relation “a student takes a course” is implemented using:

the ***addCourse*** **method** in the Student class the

addStuent method in the Course class

The relation “a faculty teaches a course” is implemented using:

the ***addCourse method*** in the Faculty class the

setFaculty method in the Course class

Aggregation and Composition

- **Aggregation** is a special form of association that represents an **ownership relationship between two objects.**

  Aggregation models **has-a** relationships

  The owner object is called an **aggregating object, and its class is called an aggregating class**

  The subject object is called an **aggregated object**, **and its class is called an aggregated class**.

  An **object can be owned by several other aggregating objects**

- If an object is **exclusively owned by an** aggregating object, the relationship between the object and its aggregating object is referred to as a composition

Examples(Aggregation and Composition)

- For example, “a student has a name” is a composition relationship between the
- Student class and the Name class.

•whereas “a student has an address” is an aggregation relationship between the Student class and the Address class, since an **address can be shared by several students**

Examples(Aggregation and Composition)

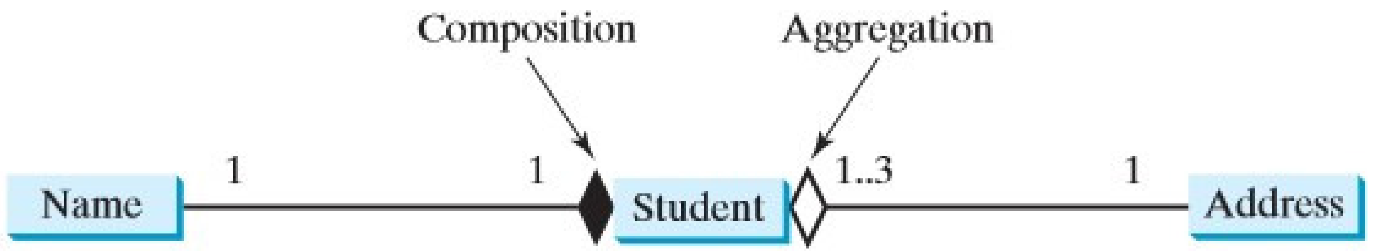
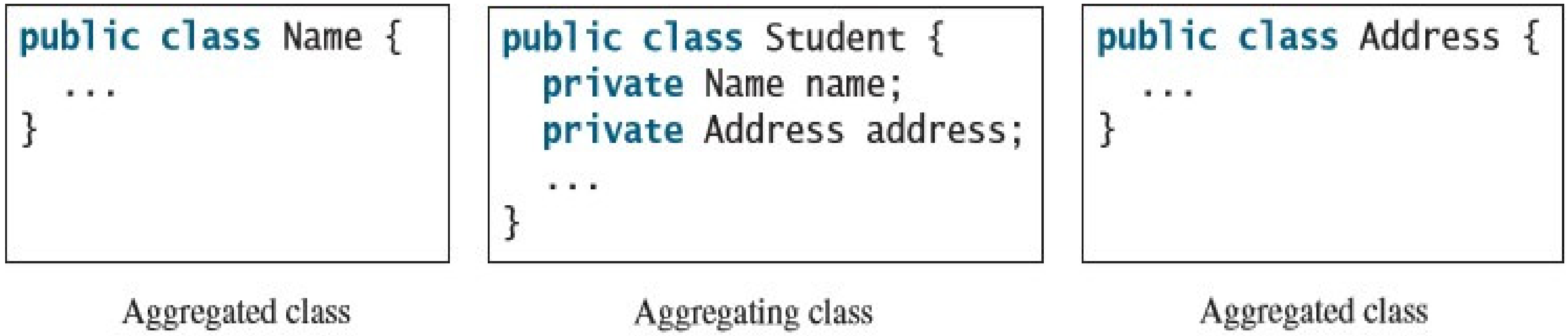


FIGURE 10.6 Each student has a name and an address.

a **filled diamond** is attached to an aggregating class (in this case, Student) to denote the composition relationship with an aggregated class (Name).

an **empty diamond** is attached to an aggregating class (Student) to denote the aggregation relationship with an aggregated class (Address).

Aggregation and Composition in Java code



```
public class Name {
  ...
}
```

Aggregated class

```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

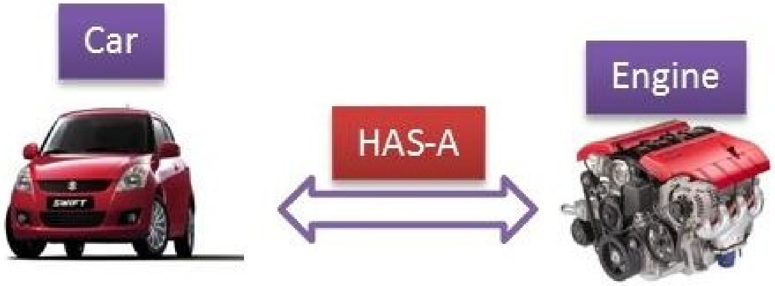
Aggregating class

```
public class Address {
  ...
}
```

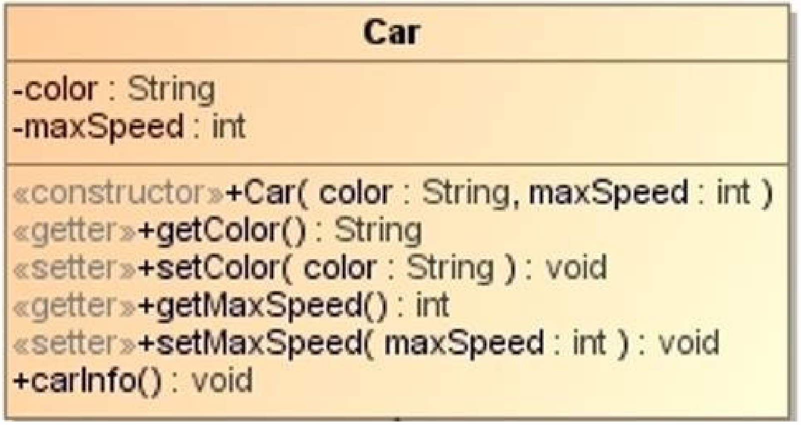
Aggregated class

FIGURE 10.7 The composition relations are implemented using data fields in classes.

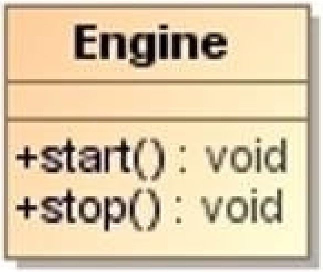
Examples (Car & Engine)



Composition (engine just for one car)

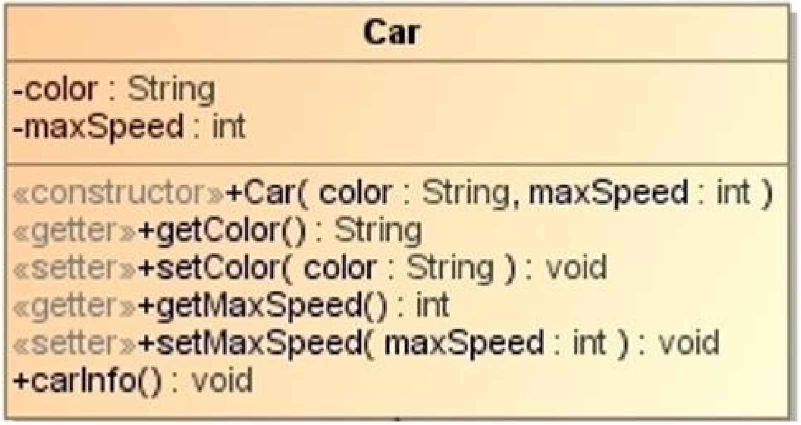


```
Car
-color : String
-maxSpeed : int
«constructor»+Car( color : String, maxSpeed : int )
«getter»+getColor() : String
«setter»+setColor( color : String ) : void
«getter»+getMaxSpeed() : int
«setter»+setMaxSpeed( maxSpeed : int ) : void
+carInfo() : void
```



```
Engine
+start() : void
+stop() : void
```

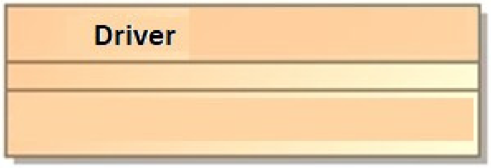
Examples (Car & Driver)



Car

-color : String
-maxSpeed : int

«constructor»+Car(color : String, maxSpeed : int)
«getter»+getColor() : String
«setter»+setColor(color : String) : void
«getter»+getMaxSpeed() : int
«setter»+setMaxSpeed(maxSpeed : int) : void
+carInfo() : void



Driver

Aggregation (shared between more than one driver)

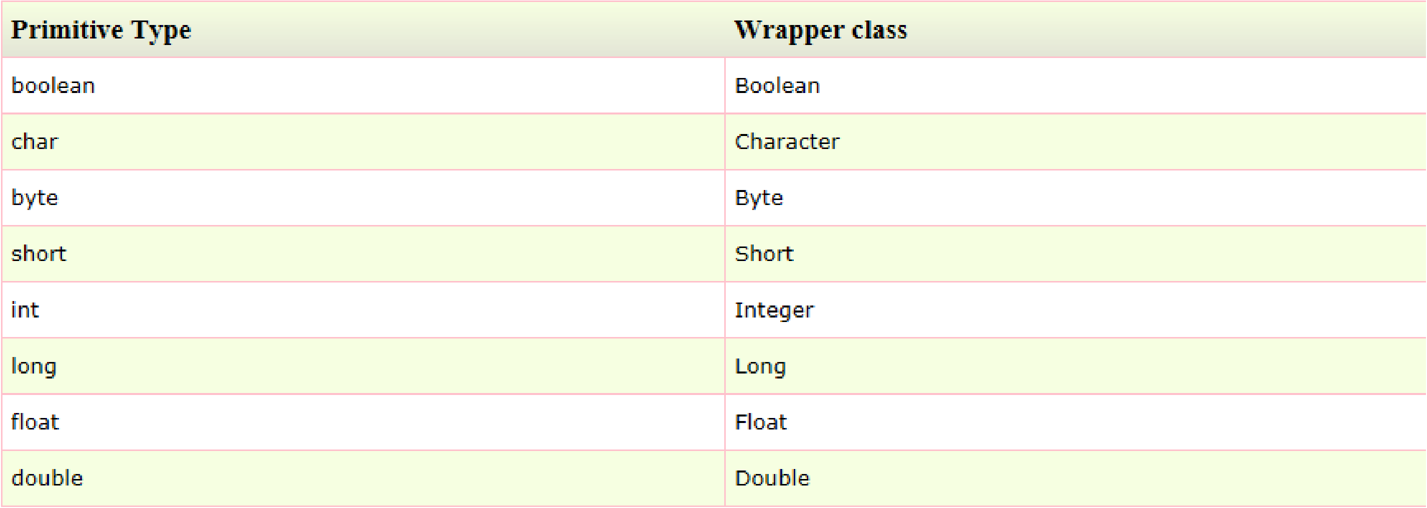
Wrapper class in java

provides the mechanism *to convert primitive into object and object into primitive*

autoboxing and **unboxing** feature converts primitive into object and object into primitive automatically. The automatic conversion of primitive into object is known as autoboxing and vice-versa unboxing

Wrapper class in java

The eight classes of *java.lang* package are known as wrapper classes in java. The list of eight wrapper classes are given below



Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

These classes are called *wrapper classes* because each wraps or encapsulates a primitive type value in an object.

Wrapper Classes

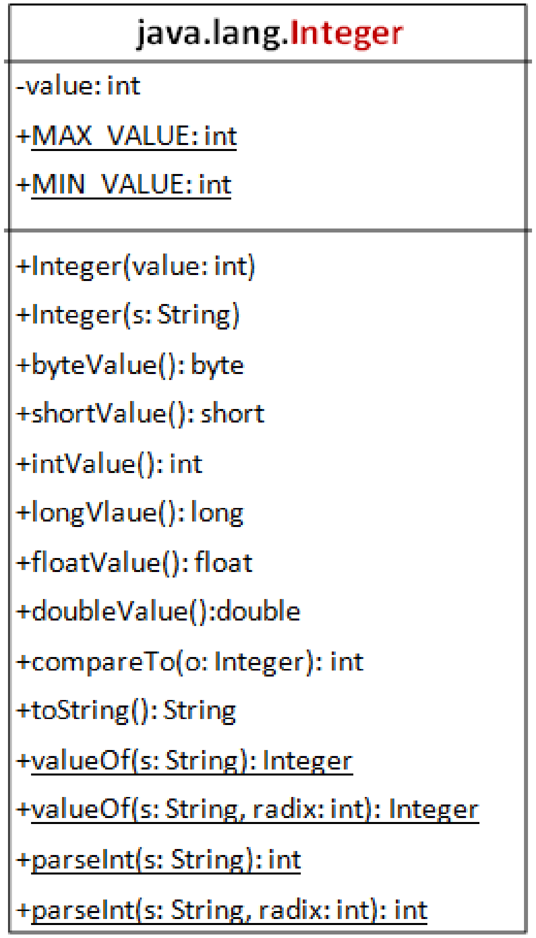
- **Boolean**
- **Character**
- **Short**
- **Byte**
- **Integer**
- **Long**
- **Float**
- **Double**

NOTE:

(1) The wrapper classes **do not** have **no-arg** constructors.

(2) The instances of all wrapper classes are **immutable**, i.e., their internal values cannot be changed once the objects are created.

The Integer and Double Classes



java.lang.Integer

-value: int

+MAX_VALUE: int

+MIN_VALUE: int

+Integer(value: int)

+Integer(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Integer): int

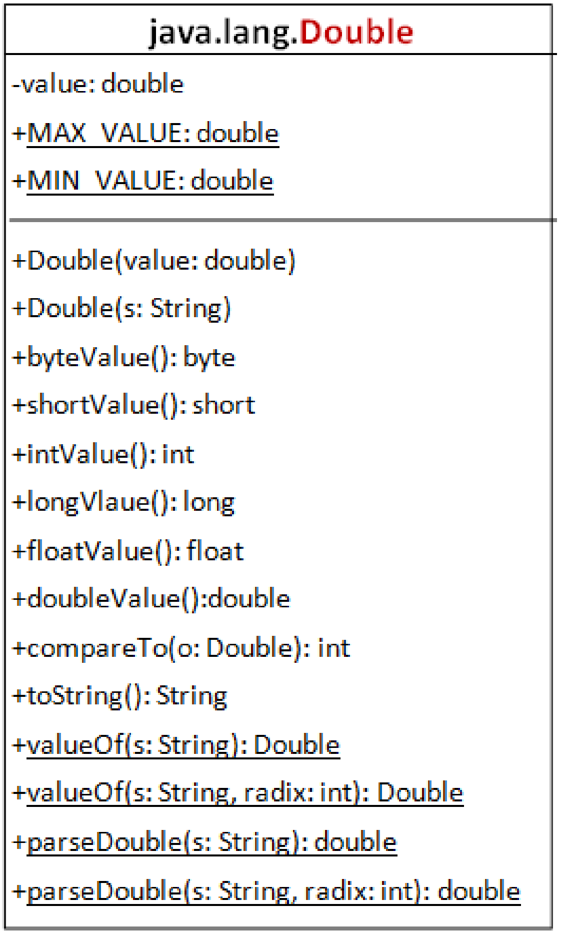
+toString(): String

+valueOf(s: String): Integer

+valueOf(s: String, radix: int): Integer

+parseInt(s: String): int

+parseInt(s: String, radix: int): int



java.lang.Double

-value: double

+MAX_VALUE: double

+MIN_VALUE: double

+Double(value: double)

+Double(s: String)

+byteValue(): byte

+shortValue(): short

+intValue(): int

+longVlaue(): long

+floatValue(): float

+doubleValue():double

+compareTo(o: Double): int

+toString(): String

+valueOf(s: String): Double

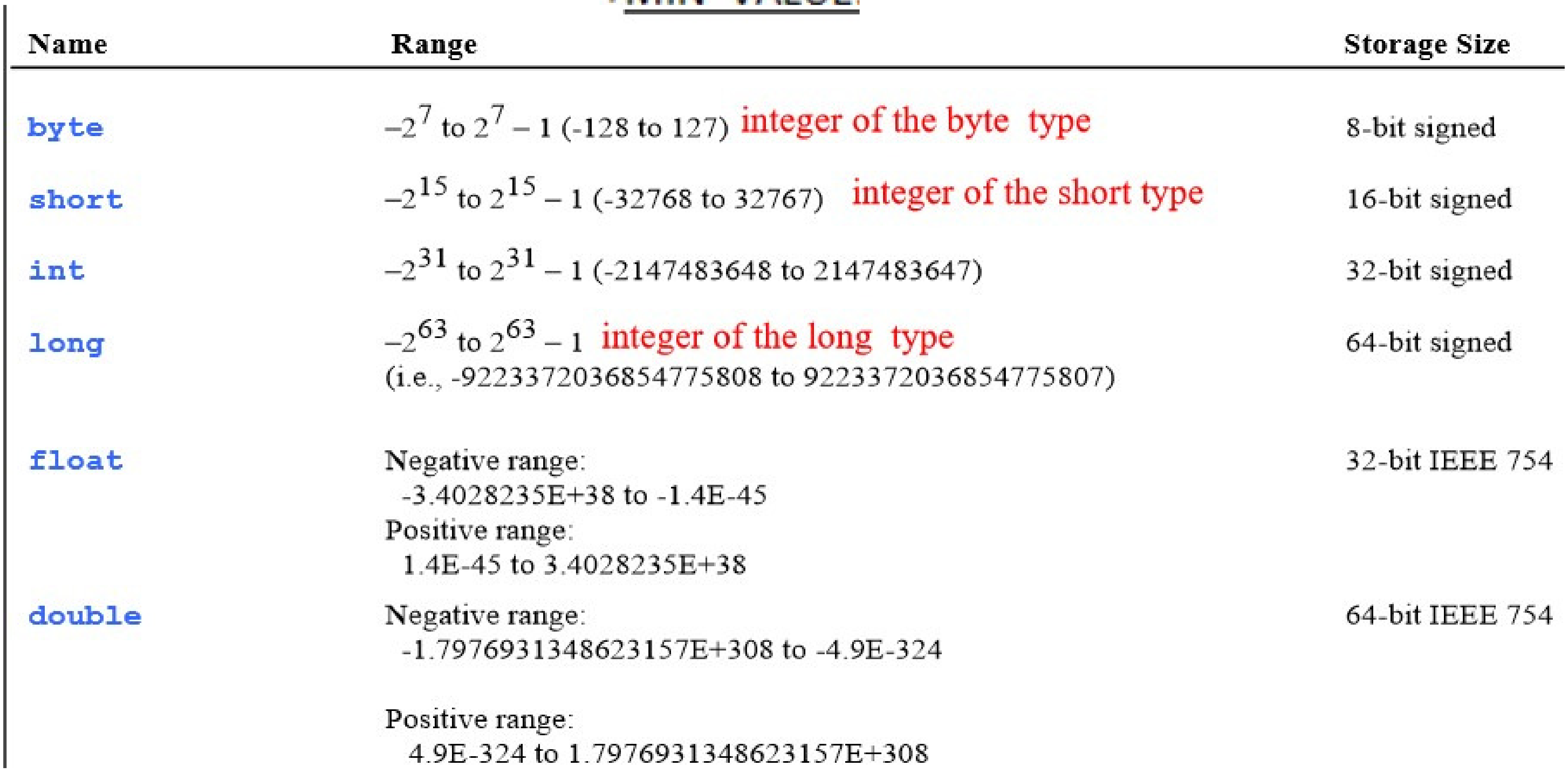
+valueOf(s: String, radix: int): Double

+parseDouble(s: String): double

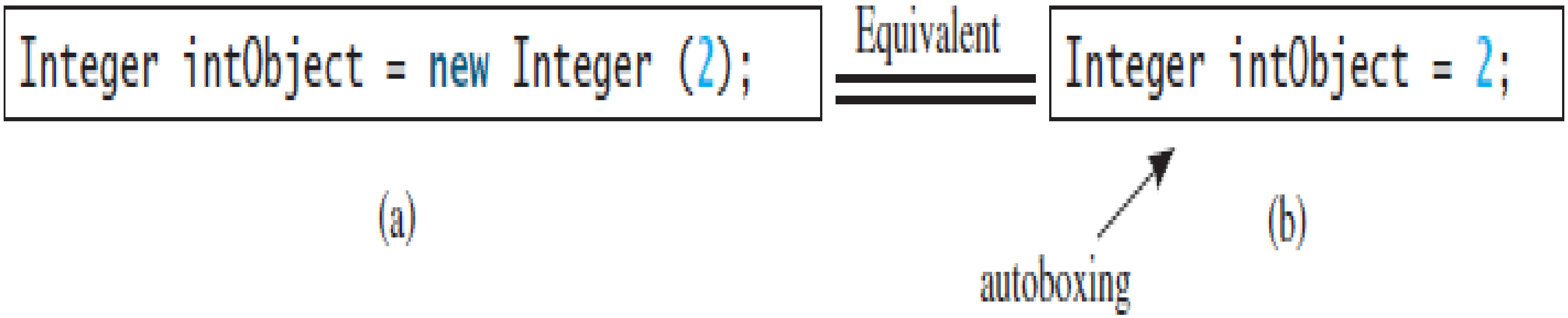
+parseDouble(s: String, radix: int): double

+MAX_VALUE

+MIN_VALUE:



Name	Range	Storage Size
`byte`	-2^{7} to $2^{7} - 1$ (-128 to 127) integer of the byte type	8-bit signed
`short`	-2^{15} to $2^{15} - 1$ (-32768 to 32767) integer of the short type	16-bit signed
`int`	-2^{31} to $2^{31} - 1$ (-2147483648 to 2147483647)	32-bit signed
`long`	-2^{63} to $2^{63} - 1$ integer of the long type (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
`float`	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
`double`	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754



```
Integer intObject = new Integer (2);
```

Equivalent

```
Integer intObject = 2;
```

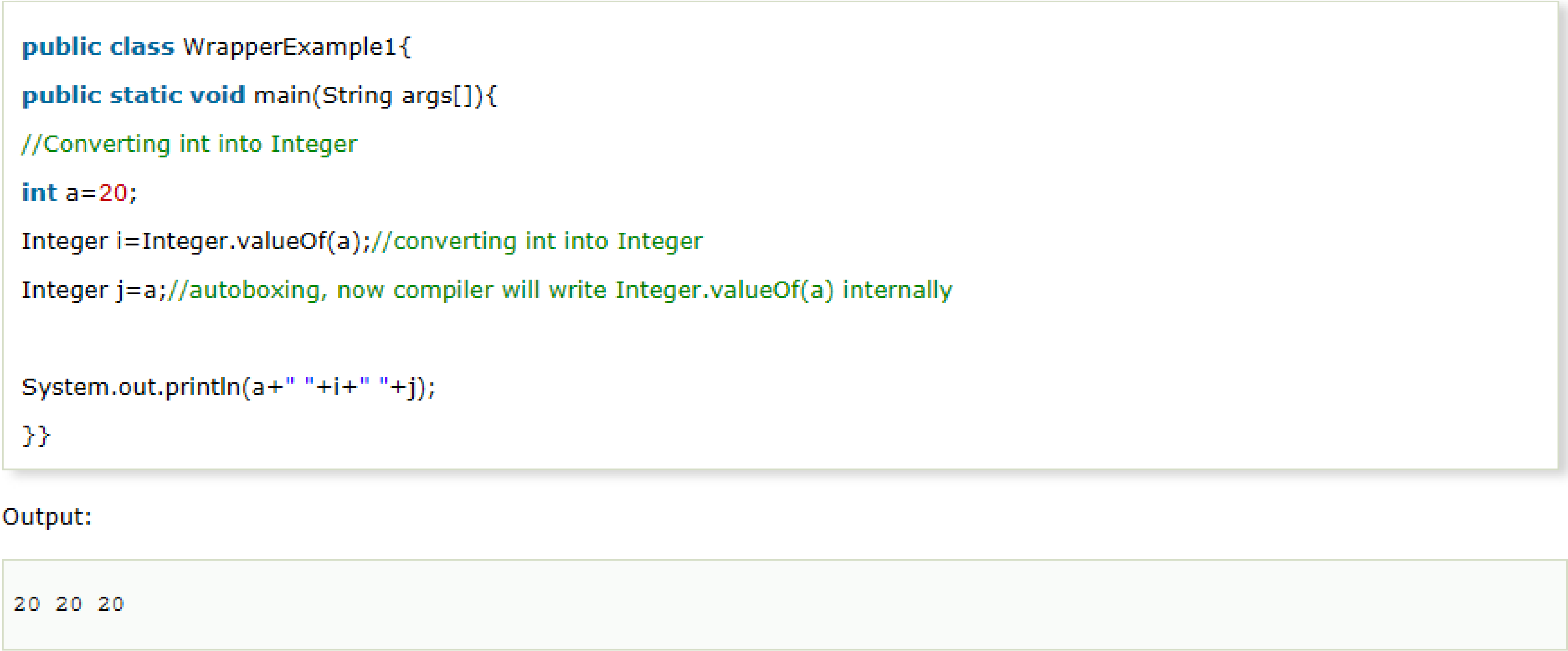
(a)

(b)

autoboxing

Examples

Wrapper class Example: Primitive to Wrapper



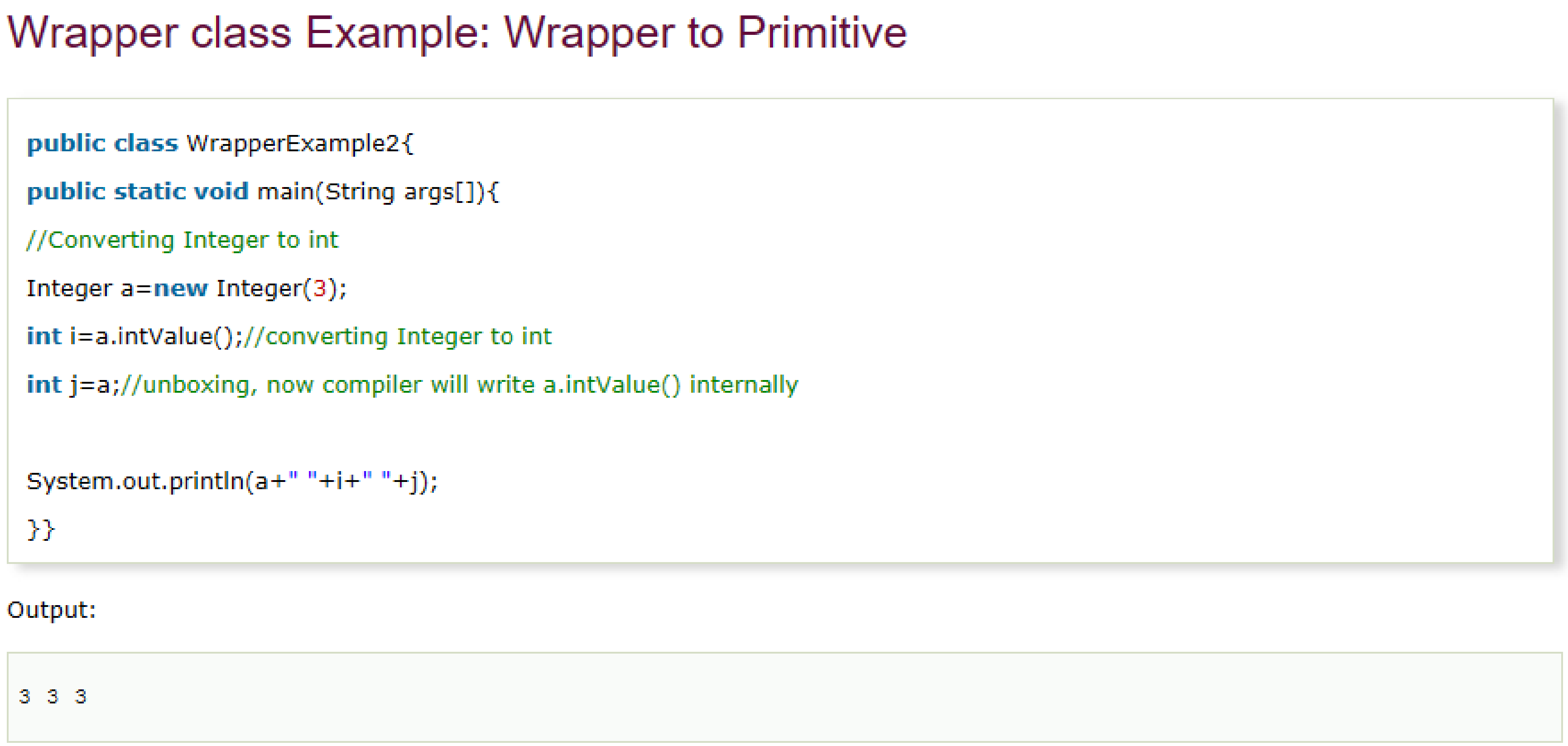
```
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
20 20 20
```

Examples



Wrapper class Example: Wrapper to Primitive

```
public class WrapperExample2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(3);
int i=a.intValue();//converting Integer to int
int j=a;//unboxing, now compiler will write a.intValue() internally

System.out.println(a+" "+i+" "+j);
}}
```

Output:

```
3 3 3
```

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants **MAX_VALUE** and **MIN_VALUE**.

MAX_VALUE represents the maximum value of the corresponding primitive data type. For **Byte**, **Short**, **Integer**, and **Long**, **MIN_VALUE** represents the minimum **byte**, **short**, **int**, and **long** values.

For **Float** and **Double**, **MIN_VALUE** represents the minimum *positive* **float** and **double** values.

Conversion Methods

Each numeric wrapper class implements the abstract methods **doubleValue**, **floatValue**, **intValue**, **longValue**, and **shortValue**, which are defined in the **Number** class.

These methods "**convert**" objects into primitive type values.

The Static valueOf Methods

The numeric wrapper classes have a useful class method, **valueOf(String s)**.

This method creates a new object initialized to the value represented by the specified string.

For example:

```
Double doubleObject = Double.valueOf("12.4");

Integer integerObject = Integer.valueOf("12");
```

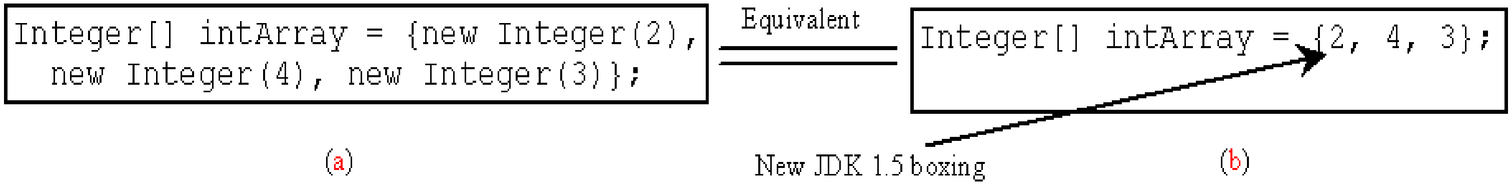
The Methods for Parsing Strings into Numbers

You have used the **parseInt** method in the **Integer** class to parse a numeric string into an **int** value and the **parseDouble** method in the **Double** class to parse a numeric string into a **double** value.

Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For
example, the following statement in (a) can be simplified as in (b):



```
Integer[] intArray = {new Integer(2),
  new Integer(4), new Integer(3)};
```

```
Integer[] intArray = {2, 4, 3};
```

```
Integer[] arr = {1, 2, 3};

System.out.println(arr[0] + arr[1] + arr[2]);
```

Unboxing

BigInteger and BigDecimal

If you need to compute with **very large integers** or **high precision floating-point** values, you can use the **BigInteger** and **BigDecimal** classes in the **java.math** package.

Both are ***immutable***.

BigInteger and BigDecimal

```
BigInteger a = new BigInteger("9223372036854775807");
BigInteger b = new BigInteger("2");
BigInteger c = a.multiply(b); // 9223372036854775807 * 2
System.out.println(c);
```

```
BigDecimal a = new BigDecimal(1.0);
BigDecimal b = new BigDecimal(3);
BigDecimal c = a.divide(b);
System.out.println(c);
```