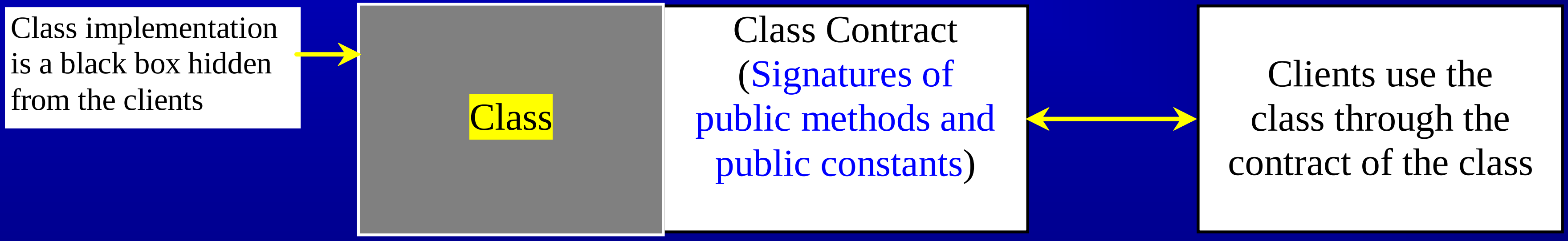
Chapter 10

Object-Oriented Thinking

Class Abstraction and Encapsulation

Class abstraction is the separation of class implementation details from users of the class. The class creator provides a description of the class to let users know how to use the class. Users of the class do not need to know the class implementation details. Thus, implementation details are encapsulated (*hidden*) from the users.



Class implementation is 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

Visibility Modifiers and Abstraction

	`public`	`private`
Variables	**Violate encapsulation**	**Enforce encapsulation**
Methods	**Provide services to clients of the class**	**Support other methods in the class**

Designing Class *Loan*

Problem Statement: A *Loan* is characterized by:
- borrowed amount *(variable)*
- interest rate *(variable)*
- start date *(variable)*
- loan duration (years) *(variable)*
- monthly payment (need to be computed) *(method)*
- total payment (need to be computed) *(method)*

Each real-life loan is a loan object with specific values for those characteristics. (e.g., car loan, mortgage, personal loan, etc.)

To program the *Loan* concept, we need to define class **Loan** with data fields (variables/attributes) and methods; and with encapsulation in mind.

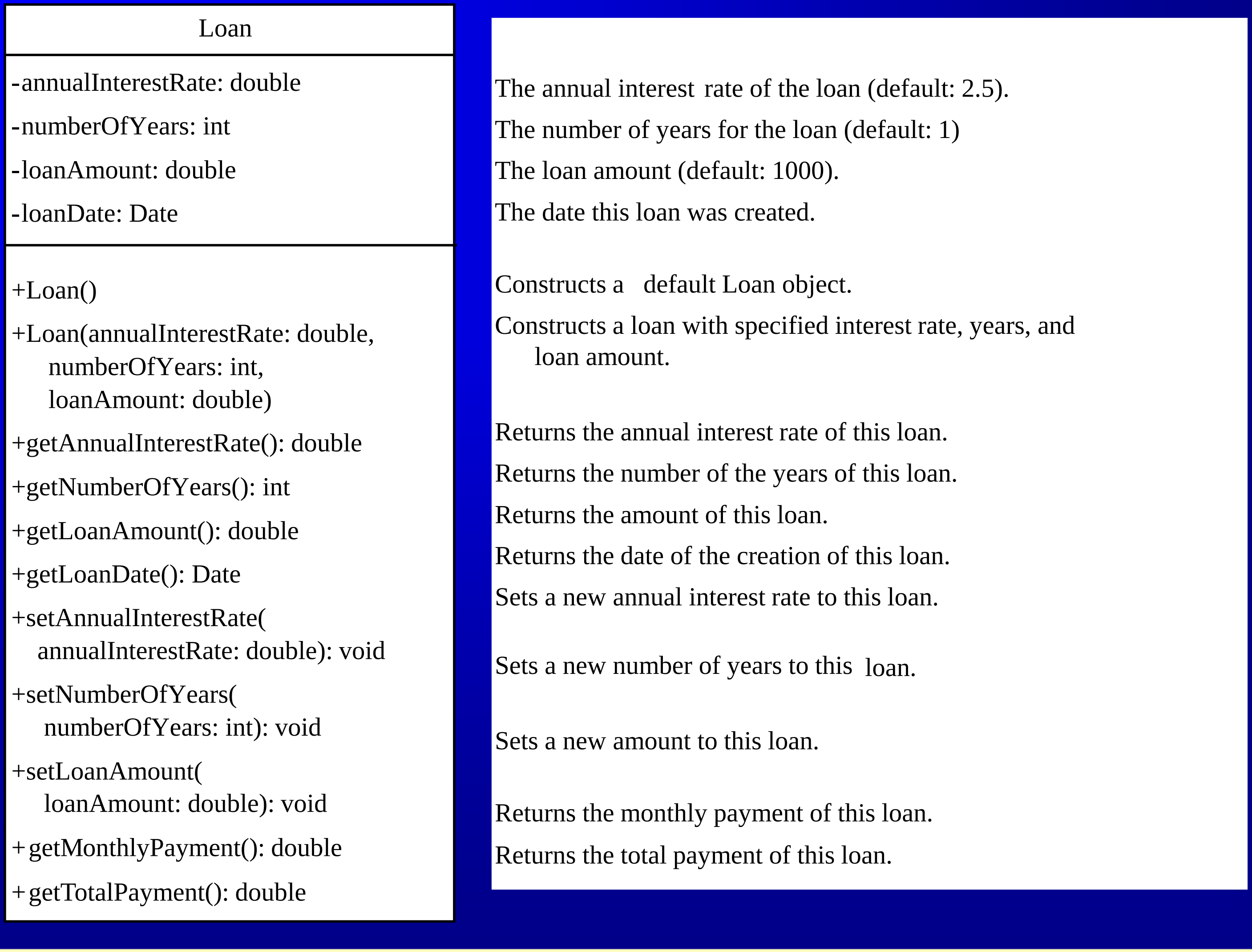
Designing the *Loan* Class

To achieve encapsulation in the class design, we need the following:

1. Define all variables to be private. No exceptions!
2. Define public methods (getters and/or setters) for each private variable that users of the class **need** to access.
3. Methods that users of the class need to know about (make use of) must be defined as public.
4. Support methods must be defined as private.

Note: All class methods have access to all of its variables.

UML modeling of class *Loan*



Loan	
-annualInterestRate: double	The annual interest rate of the loan (default: 2.5).
-numberOfYears: int	The number of years for the loan (default: 1)
-loanAmount: double	The loan amount (default: 1000).
-loanDate: Date	The date this loan was created.
+Loan()	Constructs a default Loan object.
+Loan(annualInterestRate: double, numberOfYears: int, loanAmount: double)	Constructs a loan with specified interest rate, years, and loan amount.
+getAnnualInterestRate(): double	Returns the annual interest rate of this loan.
+getNumberOfYears(): int	Returns the number of the years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate to this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years to this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount to this loan.
+getMonthlyPayment(): double	Returns the monthly payment of this loan.
+getTotalPayment(): double	Returns the total payment of this loan.

Class *Loan* and class *TestLoanClass* start on page 367.

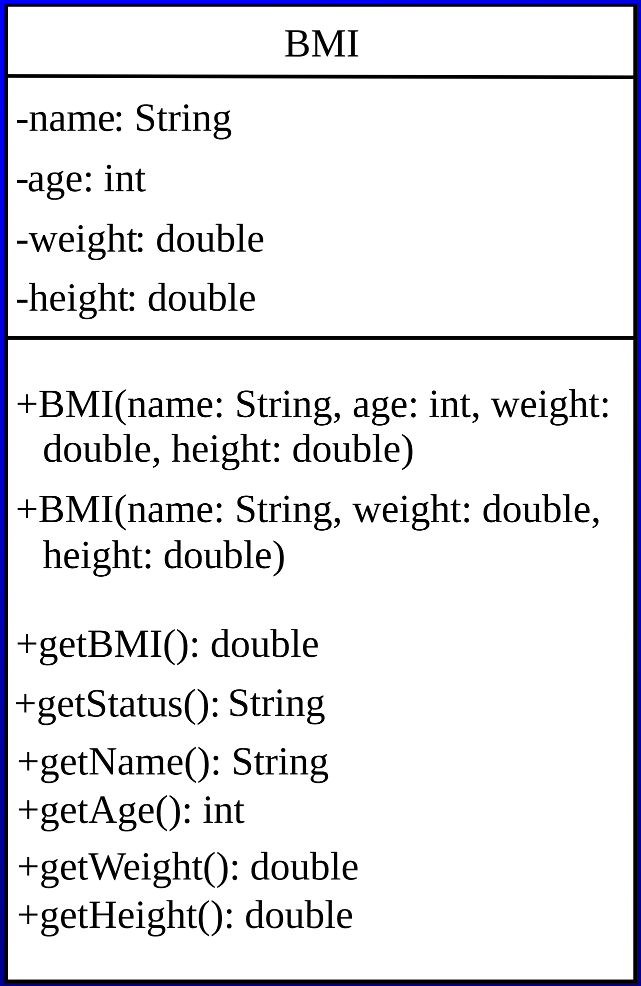
Class *Loan* Constructor Methods

```
// see complete class code on page 368

// Default constructor with default values
public Loan() {
  this(2.5, 1, 1000);  // calls the second constructor to create
                       // a loan object with default values.
                       // This is same as:
                       // annualInterestRate = 2.5;
                       // numberOfYears = 1;
                       // loanAmount = 1000;

// Construct a loan with specified rate, number of years, and amount
public Loan(double annualInterestRate, int numberOfYears, double
            loanAmount) {
  this.annualInterestRate = annualInterestRate;
  this.numberOfYears = numberOfYears;
  this.loanAmount = loanAmount;
  loanDate = new java.util.Date();  // creates date object
}
```

Class *TestLoanClass*

```
import java.util.Scanner;

public class TestLoanClass {
    public static void main(String[] args) { // Main method
    Scanner input = new Scanner(System.in);  // Create a Scanner
    // Enter yearly interest rate
    System.out.print("Enter yearly interest rate, for example, 8.25: ");
    double annualInterestRate = input.nextDouble();
    // Enter number of years
    System.out.print("Enter number of years as an integer: ");
    int numberOfYears = input.nextInt();
    // Enter loan amount
    System.out.print("Enter loan amount, for example, 120000.95: ");
    double loanAmount =  input.nextDouble();
    // Create Loan object
    Loan loan = new Loan(annualInterestRate, numberOfYears, loanAmount);
    // Display loan date, monthly payment, and total payment
    System.out.println(
     "The was loan created on: " + loan.getLoanDate().toString() + "\n" +
     "The monthly payment is:  " + loan.getMonthlyPayment() + "\n" +
     "The total payment is:    " + loan.getTotalPayment());
  }
}
```

Another OO Example: Class BMI



BMI	
-name: String	The name of the person.
-age: int	The age of the person.
-weight: double	The weight of the person in pounds.
-height: double	The height of the person in inches.
+BMI(name: String, age: int, weight: double, height: double)	Creates a BMI object with the specified name, age, weight, and height.
+BMI(name: String, weight: double, height: double)	Creates a BMI object with the specified name, weight, height, and a default age 20
+getBMI(): double	Returns the BMI
+getStatus(): String	Returns the BMI status (e.g., normal, overweight, etc.)
+getName(): String	Return name
+getAge(): int	Return age
+getWeight(): double	Return weight
+getHeight(): double	Return height

Class BMI

```
public class BMI {
  private String name;
  private int age;
  private double weight; // in pounds
  private double height; // in inches
  public static final double KILOGRAMS_PER_POUND = 0.45359237;
  public static final double METERS_PER_INCH = 0.0254;

  // constructors
  public BMI(String name, int age, double weight, double height) {
    this.name = name;
    this.age = age;
    this.weight = weight;
    this.height = height;  }

  public BMI(String name, double weight, double height) {
    this(name, 20, weight, height);
}
  // getters
  public String getName() { return name; }
  public int getAge() { return age; }
  public double getWeight() { return weight; }
  public double getHeight() { return height; }

// continue next slide
```

```
this.name = name;
this.age = 20;
this.weight = weight;
this.height = height;
```

Class BMI

```
  // compute PMI
  public double getBMI() {

    double bmi = weight * KILOGRAMS_PER_POUND /
      ((height * METERS_PER_INCH) * (height * METERS_PER_INCH));
    return Math.round(bmi * 100) / 100.0;
  }

  // determine status
  public String getStatus() {

    double bmi = getBMI();

    if (bmi < 18.5)
      return "Underweight";
    else if (bmi < 25)
      return "Normal";
    else if (bmi < 30)
      return "Overweight";
    else
      return "Obese";
  }
}
```

The Test Program

```
public class UseBMIClass {
  public static void main(String[] args) {

    BMI bmi1 = new BMI("John Doe", 18, 145, 70);

    System.out.println("The BMI for " + bmi1.getName() + " is "
      + bmi1.getBMI() + " " + bmi1.getStatus());

    BMI bmi2 = new BMI("Peter King", 215, 70);

    System.out.println("The BMI for " + bmi2.getName() + " is "
      + bmi2.getBMI() + " " + bmi2.getStatus());
  }
}
```

```
----jGRASP exec: java UseBMIClass

The BMI for John Doe is 20.81 Normal
The BMI for Peter King is 30.85 Obese

----jGRASP: operation complete.
```

Wrapper Classes

Java primitive types are NOT objects.

Often we need to treat primitive values as objects.

The solution is to convert a primitive type value, such as 45, to an object that holds value 45.

Java provides Wrapper Classes for all primitive types.

Wrapper Classes

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

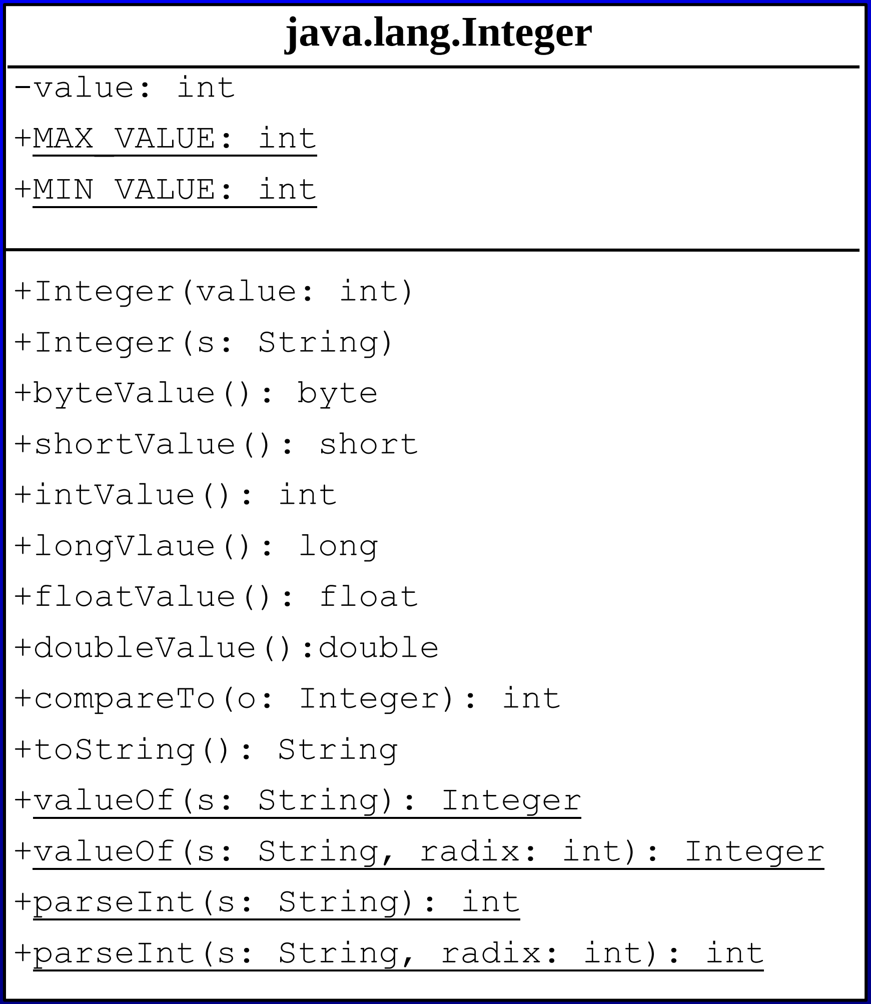
Note:

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

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

(3) A wrapper class object contains one value of the class type.

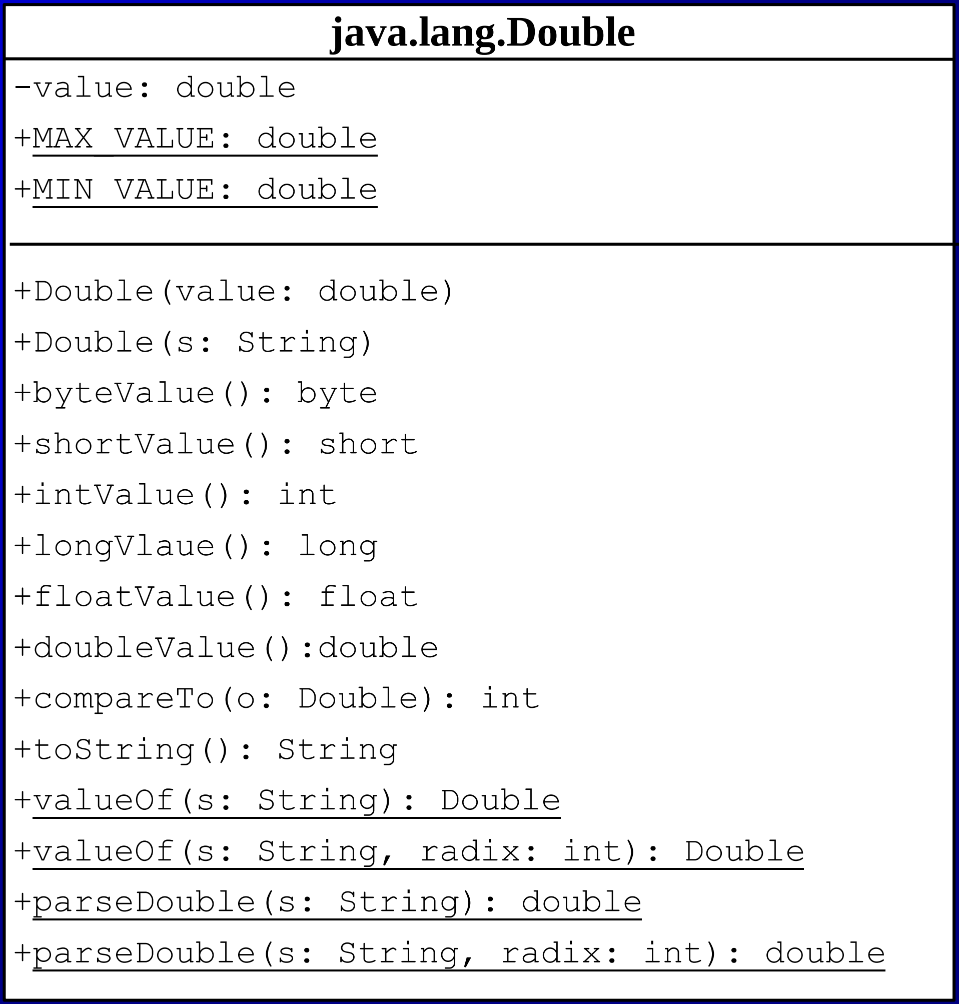
The Integer and Double Classes



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
```

Numeric Wrapper Class Constructors

We can construct a wrapper object either from:

1) primitive data type value

2) string representing the numeric value

The constructors for classes Integer and Double are:

```
public Integer(int value)
public Integer(String s)
public Double(double value)
public Double(String s)
```

Examples:

```
    Integer intObject1    = new Integer(90);
    Integer intObject2    = new Integer("90");
    Double  doubleObject1 = new Double(95.7);
    Double  doubleObject2 = new Double("95.7");
// Similar syntax for Float, Byte, Short, and Long types.
```

Numeric Wrapper Class Constants

Each numerical wrapper class has 2 constants:

`MAX_VALUE`: represents the maximum value of the type.

`MIN_VALUE`: represents the minimum value of the type.

Examples:

```
System.out.println("Max integer is: " + Integer.MAX_VALUE);
System.out.println("Min integer is: " + Integer.MIN_VALUE);
System.out.println("Max float is: " + Float.MAX_VALUE);
System.out.println("Min float is: " + Float.MIN_VALUE);
System.out.println("Max short is: " + Short.MAX_VALUE);
System.out.println("Min short is: " + Short.MIN_VALUE);
System.out.println("Max byte is: " + Byte.MAX_VALUE);
System.out.println("Min byte is: " + Byte.MIN_VALUE);
```

Conversion Methods

Each numeric wrapper class implements conversion methods that convert an object of a wrapper class to a primitive type:

```
doubleValue(), floatValue(), intValue()
longValue(), and shortValue().
```

Examples:

```
Double myValue = new Double(97.50);
System.out.println(myValue.intValue());   //gives 97
System.out.println(myValue.floatValue()); //gives 97.5
System.out.println(myValue.shortValue()); //gives 97
System.out.println(myValue.longValue());  //gives 97
```

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.

Examples:

```
    Double doubleObject = Double.valueOf("95.79");
    Integer integerObject = Integer.valueOf("86");
    Float floatObject = Float.valueOf("95.54");
    Long longObject = Long.valueOf("123456789");
    Short shortObject = Short.valueOf("123");
    Byte byteObject = Byte.valueOf("12");
```

Methods for Parsing Strings into Numbers

Parsing methods allow us to pars numeric strings into numeric types. Each numeric wrapper class has two overloaded parsing methods:

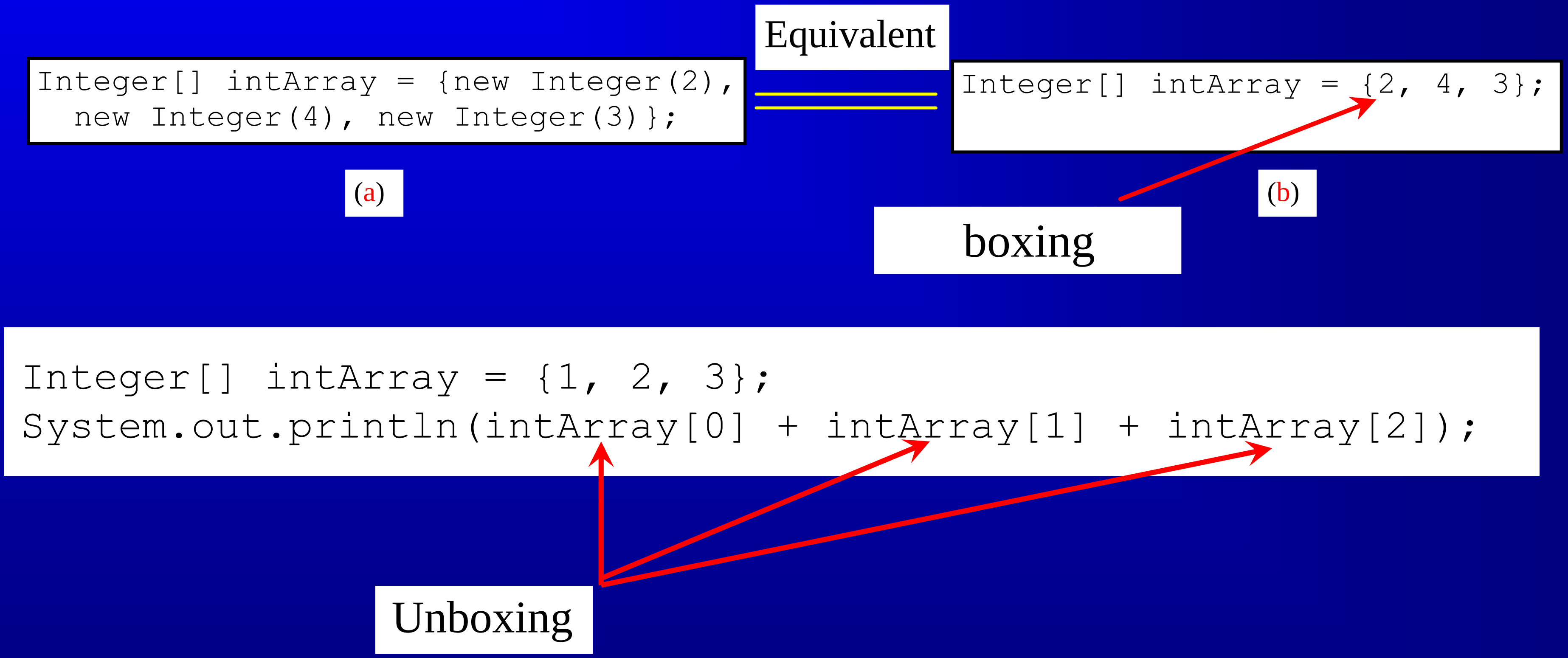
```
Public static int parseInt(String s)
Public static int parseInt(String s, int radix)
```

Examples:

```
    int A = Integer.parseInt("25");    //A has 25
    System.out.println(A);
    int B = Integer.parseInt("110",2); //B has 6
    System.out.println(B);
    int C = Integer.parseInt("25",8);  //C has 21
    System.out.println(C);
    int D = Integer.parseInt("25",10); //D has 25
    System.out.println(D);
    int E = Integer.parseInt("25",16); //E has 37
    System.out.println(E);
```

Automatic Conversion

Java allows primitive type and wrapper classes to be converted automatically.



Equivalent

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

(a)

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

(b)

boxing

```
Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);
```

Unboxing

BigInteger and BigDecimal Classes

To work with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package.

Examples:

```
 BigInteger bigA = new BigInteger("12345678923456789");
 BigInteger bigB = new BigInteger("7");
 BigDecimal bigC = new BigDecimal("1245.56789");
 BigDecimal bigD = new BigDecimal("2");
 System.out.println(bigA.multiply(bigB));
 System.out.println(bigC.divide(bigD, 20,BigDecimal.ROUND_UP));
```

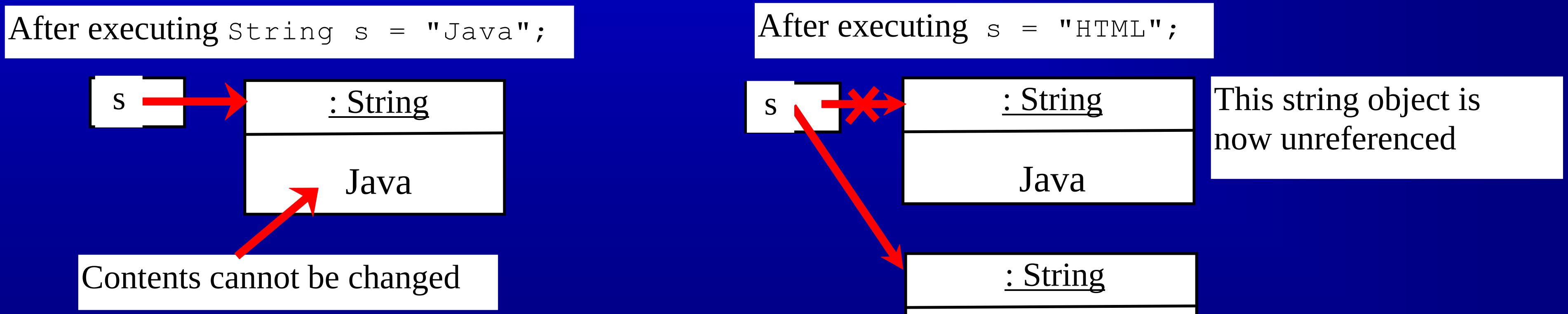
```
The output is:
       86419752464197523
       622.78394500000000000000
```

The `String` Class Revisited

A String object is immutable; its contents cannot be changed.

The following code does NOT change the content of string s.

```
String s = "Java";
s = "HTML";
```



After executing `String s = "Java";`

Contents cannot be changed

After executing `s = "HTML";`

This string object is now unreferenced

End of Chapter 10