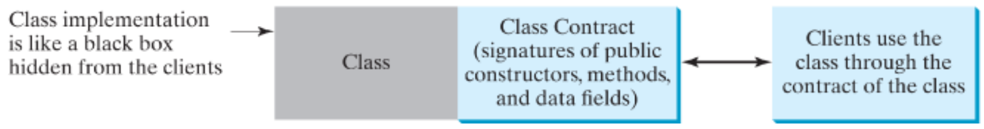
Note #2 – OO Design

- **Class Abstractions and Encapsulation**

Class abstraction is separation of class implementation from the use of a class. The details of implementation are encapsulated and hidden from the user. This is known as class encapsulation, which separates class implementation from how the class is used.


Class implementation is like a black box hidden from the clients
Class
Class Contract (signatures of public constructors, methods, and data fields)
Clients use the class through the contract of the class

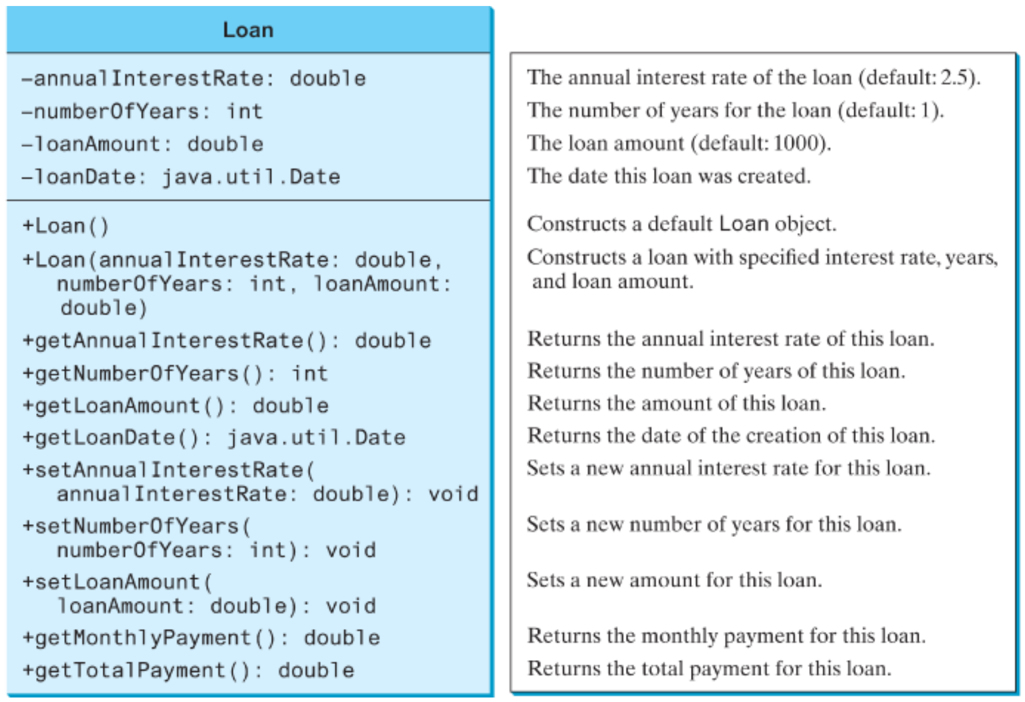
Method abstraction is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is also known as **information hiding** or **encapsulation**. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a "black box."

The creator of a class describes the functions of the class and lets the user know how the class can be used. The collection of public constructors, methods, and fields that are accessible from outside the class, together with the description of how these members are expected to behave, serves as the class's contract. The user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. This is called class encapsulation. For example, you can create a Circle object and find the area of the circle without knowing how the area is computed. For this reason, a class is also known as an **abstract data type** (ADT).

Class abstraction and encapsulation are two sides of the same coin. Many real-life examples illustrate the concept of class abstraction. Consider, for instance, building a computer system. Your personal computer has many components—a CPU, memory, disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, you need to know only how each component is used and how it interacts with the others. You don't need to know how the components work internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer-system analogy precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use in a computer, with properties such as fan size and speed and methods such as start and stop. A specific fan is an instance of this class with specific property values.

As another example, consider getting a loan. A specific loan can be viewed as an object of a **Loan** class. The interest rate, loan amount, and loan period are its data properties and computing the monthly and total payments are its methods. When you buy a car, a loan object is created by instantiating the class with your loan interest rate, loan amount, and loan period. You can then use the methods to find the monthly payment and total payment of your loan. As a user of the **Loan** class, you don't need to know how these methods are implemented.



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: java.util.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 years of this loan.
+getLoanAmount(): double	Returns the amount of this loan.
+getLoanDate(): java.util.Date	Returns the date of the creation of this loan.
+setAnnualInterestRate(annualInterestRate: double): void	Sets a new annual interest rate for this loan.
+setNumberOfYears(numberOfYears: int): void	Sets a new number of years for this loan.
+setLoanAmount(loanAmount: double): void	Sets a new amount for this loan.
+getMonthlyPayment(): double	Returns the monthly payment for this loan.
+getTotalPayment(): double	Returns the total payment for this loan.

From a class developer's perspective, a class is designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through constructors, properties, and methods.

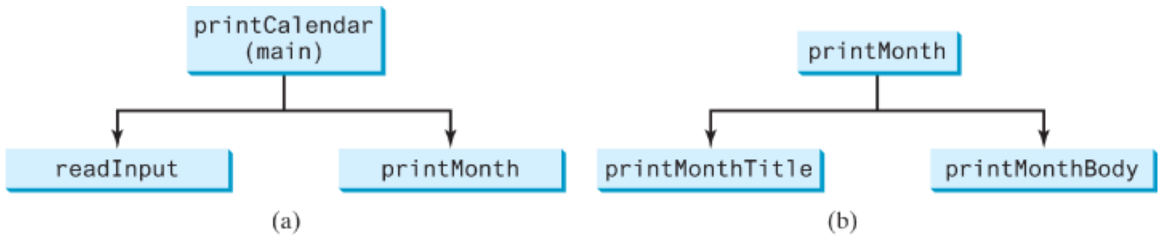
- **Object Oriented Thinking**

The procedural paradigm focuses on designing methods. The object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. Classes provide more flexibility and modularity for building reusable software. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

- **Stepwise Refinement**

Developing software is essentially a problem-solving process. How would you get started on this process? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem-solving flows as smoothly as possible, for example, you begin by using method abstraction to isolate details from design and only later implements the details. Stepwise refinement breaks a large problem into smaller manageable subproblems. Each subproblem can be implemented using a method. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

As an example, in the figure shown below, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem

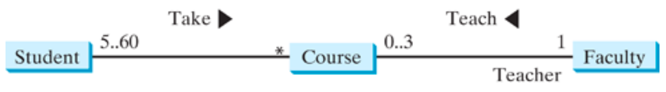


- **Class Relationships**

To design classes, you need to explore the relationships among classes. The common relationships among classes are association, aggregation, composition, and inheritance.

   - ***Association***

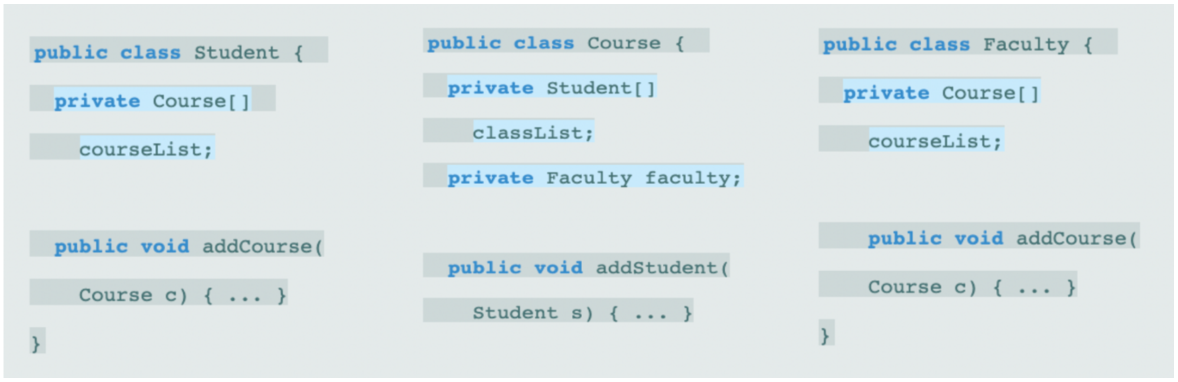
Association is a general binary relationship that describes an activity between two classes. For example, a student taking a course is an association between the Student class and the Course class, and a faculty member teaching a course is an association between the Faculty class and the Course class. These associations can be represented in UML graphical notation below.



An association is illustrated by a solid line between two classes with an optional label that describes the relationship. In the figure above, the labels are "Take" and "Teach". Each relationship may have an optional small black triangle that indicates the direction of the relationship. In this figure, the ▸ indicates that a student takes a course (as opposed to a course taking a student). Each class involved in the relationship may have a role name that describes the role it plays in the relationship. In the 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. In the above figure, each student may take any number of courses, and each course must have at least 5 and at most 60 students. Each course is taught by only one faculty member, and a faculty member may teach from 0 to 3 courses per semester.

In Java code, you can implement associations by using data fields and methods. For example, the relationships in the above figure may be implemented using the classes in below. The relation "a student takes a course" is implemented using the **addCourse** method in the Student class and the **addStudent** method in the Course class. The relation "a faculty teaches a course" is implemented using the **addCourse** method in the Faculty class and the **setFaculty** method in the Course class. The Student class may use a list to store the courses that the student is taking, the Faculty class may use a list to store the courses that the faculty is teaching, and the Course class may use a list to store students enrolled in the course and a data field to store the instructor who teaches the course.



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

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

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

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

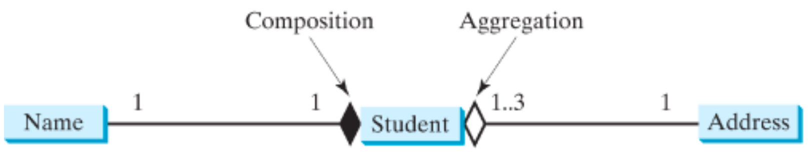
```
public class Faculty {
  private Course[]
    courseList;

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

There are many possible ways to implement relationships. For example, the student and faculty information in the Course class can be omitted, since they are already in the Student and Faculty class. Likewise, if you don't need to know the courses a student takes or a faculty member teaches, the data field **courseList** and the **addCourse** method in Student or Faculty can be omitted.

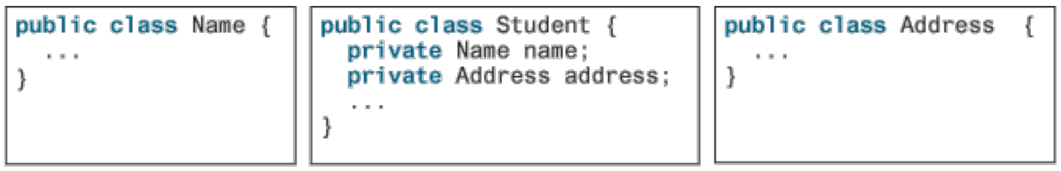
- ***Aggregation and Composition***

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.

We refer aggregation between two objects as composition if the existence of the aggregated object is dependent on the aggregating object. In other words, if a relationship is composition, the aggregated object cannot exist on its own. For example, "a student has a name" is a composition relationship between the Student class and the Name class because Name is dependent on Student, whereas "a student has an address" is an aggregation relationship between the Student class and the Address class because an address can exist by itself. Composition implies exclusive ownership. One object owns another object. When the owner object is destroyed, the dependent object is destroyed as well. In UML, a filled diamond is attached to an aggregating class (in this case, Student) to denote the composition relationship with an aggregated class (Name), and an empty diamond is attached to an aggregating class (Student) to denote the aggregation relationship with an aggregated class (Address), as shown below.



In the above figure, each student has only one multiplicity—address—and each address can be shared by up to 3 students. Each student has one name, and the name is unique for each student. An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationships in the figure above may be implemented using the classes in below. The relation "a student has a name" and "a student has an address" are implemented in the data field name and address in the Student class.



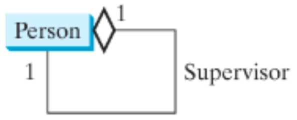
```
public class Name {
  ...
}
```

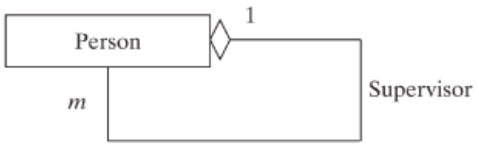
```
public class Student {
  private Name name;
  private Address address;
  ...
}
```

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

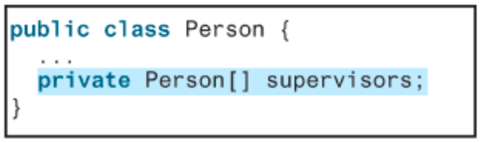
Aggregated class Aggregating class Aggregated class

Aggregation may exist between objects of the same class. For example, a person may have a supervisor. This is illustrated in the figure below. In the relationship "a person has a supervisor," a supervisor can be represented as a data field in the Person class. If a person can have several supervisors, you may use an array to store supervisor.





(a)

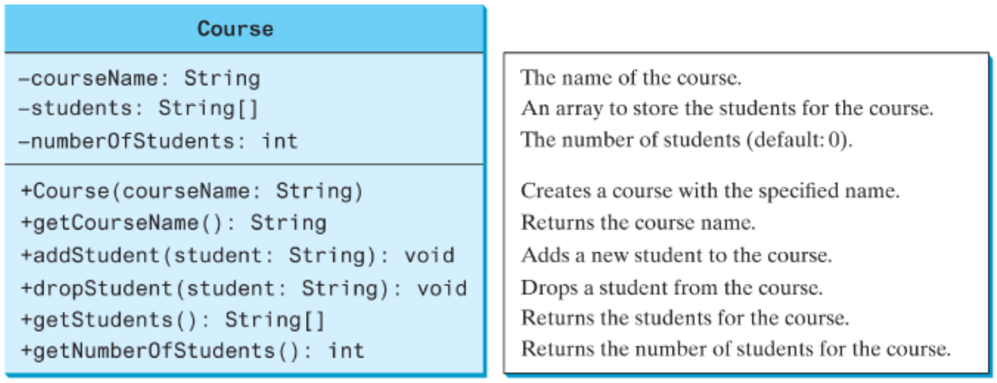


```
public class Person {
  ...
  private Person[] supervisors;
}
```

(b)

- **Example 1 – Designing the Course Class**

Suppose you need to process course information. Each course has a name and has students enrolled. You should be able to add/drop a student to/from the course. You can use a class to model the courses, as shown below. A Course object can be created using the constructor **Course(String name)** by passing a course name. You can add students to the course using the **addStudent(String student)** method, drop a student from the course using the **dropStudent(String student)** method, and return all the students in the course using the **getStudents()** method.

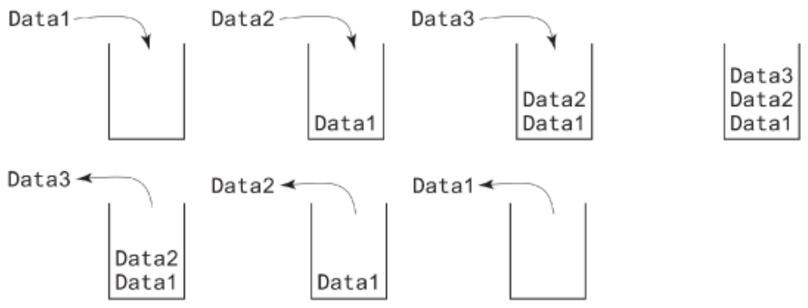


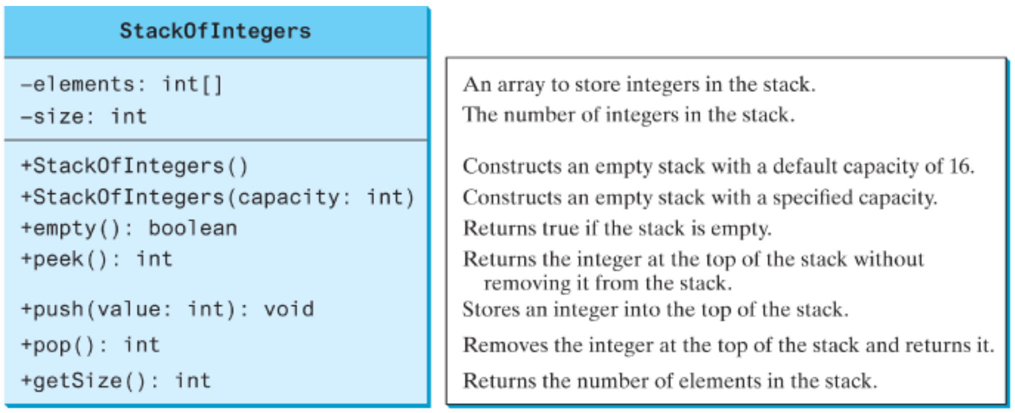
Course	
-courseName: String	The name of the course.
-students: String[]	An array to store the students for the course.
-numberOfStudents: int	The number of students (default: 0).
+Course(courseName: String)	Creates a course with the specified name.
+getCourseName(): String	Returns the course name.
+addStudent(student: String): void	Adds a new student to the course.
+dropStudent(student: String): void	Drops a student from the course.
+getStudents(): String[]	Returns the students for the course.
+getNumberOfStudents(): int	Returns the number of students for the course.

- **Example 2 – Designing the Stack Class**

Recall that a **stack** is a data structure that holds data in a last-in, first-out fashion. Stacks have many applications. For example, the compiler uses a stack to process method invocations. When a method is invoked, its parameters and local variables are pushed into a stack. When a method calls another method,

the new method's parameters and local variables are pushed into the stack. When a method finishes its work and returns to its caller, its associated space is released from the stack.
You can define a class to model stacks. For simplicity, assume the stack holds the **int** values. Thus, name the stack class **StackOfIntegers**. The UML diagram for the class is shown below.





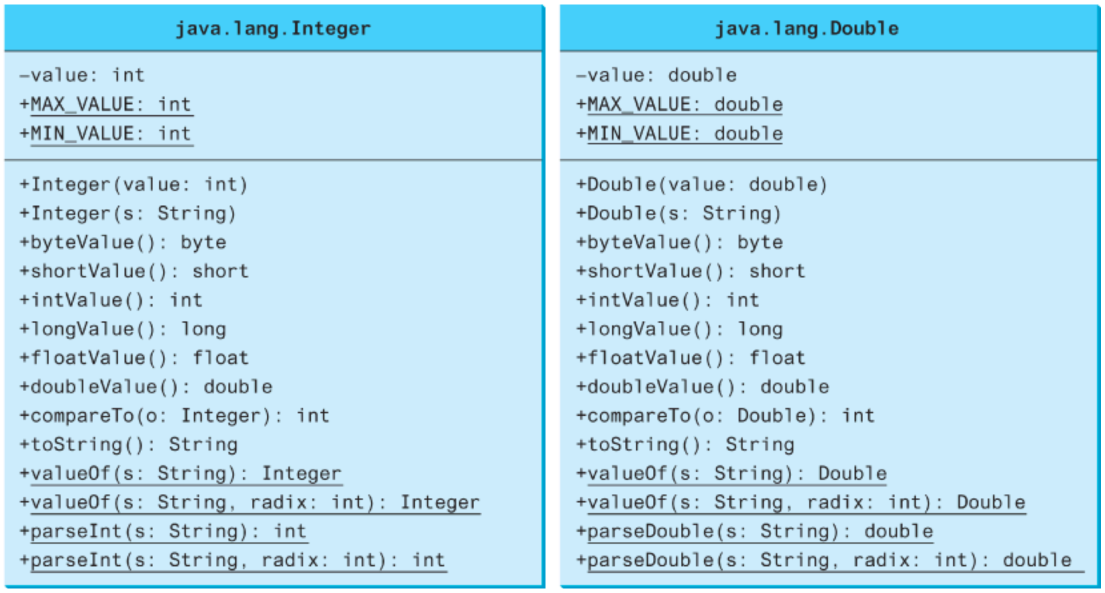
StackOfIntegers	
-elements: int[]	An array to store integers in the stack.
-size: int	The number of integers in the stack.
+StackOfIntegers()	Constructs an empty stack with a default capacity of 16.
+StackOfIntegers(capacity: int)	Constructs an empty stack with a specified capacity.
+empty(): boolean	Returns true if the stack is empty.
+peek(): int	Returns the integer at the top of the stack without removing it from the stack.
+push(value: int): void	Stores an integer into the top of the stack.
+pop(): int	Removes the integer at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

```
public class Stack {
   private Object [] elements;
   private int top;
   public Stack( int size ) {   //constructor
      elements = new Object[size];
      top = 0;
   }
   ....
}
```

- **Processing Primitive Data Type Values as Objects**

A primitive-type value is not an object, but it can be wrapped in an object using a wrapper class in the Java API. Owing to performance considerations, primitive data type values are not objects in Java. Because of the overhead of processing objects, the language's performance would be adversely affected if primitive data type values were treated as objects. However, many Java methods require the use of objects as arguments. Java offers a convenient way to incorporate, or wrap, a primitive data type value into an object (e.g., wrapping an int into an Integer object, wrapping a double into a Double object, and wrapping a char into a Character object). By using a wrapper class, you can process primitive data type values as objects. Java provides Boolean, Character, Double, Float, Byte, Short, Integer, and Long wrapper classes in the **java.lang** package for primitive data types. The Boolean class wraps a Boolean value true or false. This section uses Integer and Double as examples to introduce the numeric wrapper classes. Most wrapper class names for a primitive type are the same as the primitive data type name with the first letter capitalized. The exceptions are Integer for int and Character for char.

Numeric wrapper classes are very similar to each other. Each contains the methods doubleValue(), floatValue(), intValue(), longValue(), shortValue(), and byteValue(). These methods "convert" objects into primitive-type values. The key features of Integer and Double are shown below.



You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value—for example, **new Double(5.0)**, **new Double("5.0")**, **new Integer(5)**, and **new Integer("5")**. The wrapper classes do not have no-arg constructors. The instances of all wrapper classes are immutable; this means that, once the objects are created, their internal values cannot be changed.

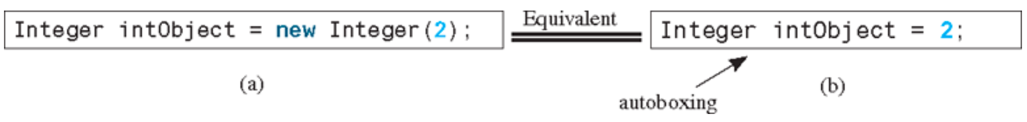
Each numeric wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. Float and Double, MIN_VALUE represents the minimum positive float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E–45), and the maximum double floating-point number (1.79769313486231570 e + 308 d) (1.79769313486231570e+308d):

The numeric wrapper classes contain the **compareTo** method for comparing two numbers and returns **1**, **0**, or **–1**, if this number is greater than, equal to, or less than the other number. For example,

```
new Double(12.4).compareTo(new Double(12.3))  returns 1 ;
new Double(12.3).compareTo(new Double(12.3))  returns 0 ;
new Double(12.3).compareTo(new Double(12.51))  returns –1 ;
```

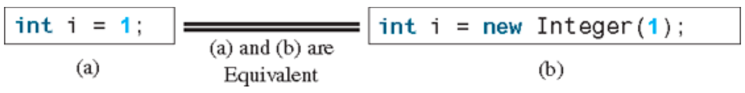
The numeric wrapper classes have a useful static method, **valueOf(String s)**. This method creates a new object initialized to the value represented by the specified string. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value based on 10 (decimal) or any specified radix (e.g., 2 for binary, 8 for octal, and 16 for hexadecimal).

A primitive-type value can be automatically converted to an object using a wrapper class, and vice versa, depending on the context. Converting a primitive value to a wrapper object is called **boxing.** The reverse conversion is called unboxing. Java allows primitive types and wrapper classes to be converted automatically. The compiler will automatically box a primitive value that appears in a context requiring an object, and unbox an object that appears in a context requiring a primitive value. This is called **autoboxing** and **autounboxing.**



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

```
Integer intObject = 2;
```



```
int i = 1;
```

```
int i = new Integer(1);
```

The BigInteger and BigDecimal classes can be used to represent integers or decimal numbers of any size and precision. 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. The largest integer of the long type is Long.MAX_VALUE (i.e., 9223372036854775807). An instance of BigInteger can represent an integer of any size. You can use new BigInteger(String) and new BigDecimal(String) to create an instance of BigInteger and BigDecimal, use the add, subtract, multiply, divide, and remainder methods to perform arithmetic operations, and use the compareTo method to compare two big numbers. For example, the following code creates two BigInteger objects and multiplies them:

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

There is no limit to the precision of a BigDecimal object. The divide method may throw an ArithmeticException if the result cannot be terminated. However, you can use the overloaded **divide(BigDecimal d, int scale, int roundingMode)** method to specify a scale and a rounding mode to avoid this exception, where scale is the maximum number of digits after the decimal point. For example, the following code creates two BigDecimal objects and performs division with scale 20 and rounding mode BigDecimal.ROUND_UP:

- **String class**

A String object is immutable; its contents cannot be changed once the string is created.

```
String newString = new String(stringLiteral);
```

The argument **stringLiteral** is a sequence of characters enclosed in double quotes. The following statement creates a **String** object **message** for the string literal **"Welcome to Java"**:

```
String message = new String("Welcome to Java");
```

Java treats a string literal as a **String** object. Thus, the following statement is valid:

```
String message = "Welcome to Java";
```

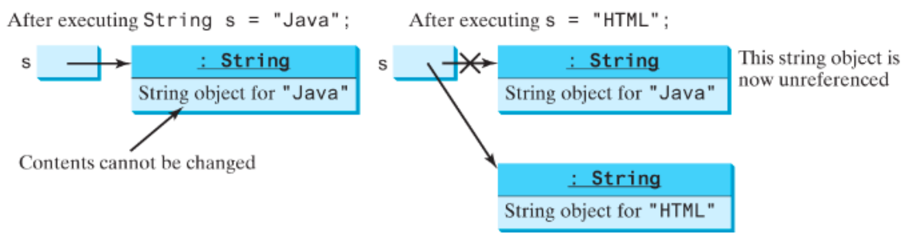
You can also create a string from an array of characters. For example, the following statements create the string **"Good Day"**:

```
char[] charArray = {'G', 'o', 'o', 'd', '', 'D', 'a', 'y'};
String message = new String(charArray);
```

A String variable holds a reference to a String object that stores a string value. Strictly speaking, the terms String variable, String object, and string value are different, but most of the time the distinctions between them can be ignored. For simplicity, the term string will often be used to refer to String variable, String object, and string value. A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

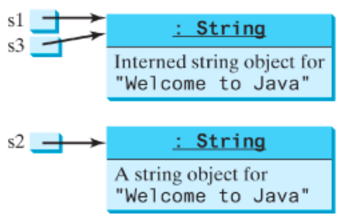
```
String s = "Java";
s = "HTML";
```

The answer is NO. The first statement creates a String object with the content "Java" and assigns its reference to s. The second statement creates a new String object with the content "HTML" and assigns its reference to s. The first String object still exists after the assignment, but it can no longer be accessed, because variable s now points to the new object, as shown below.


After executing String s = "Java";
After executing s = "HTML";
s
: String
String object for "Java"
Contents cannot be changed
s
: String
String object for "Java"
This string object is now unreferenced
: String
String object for "HTML"

Because strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called an **interned string**. For example, the following statements:

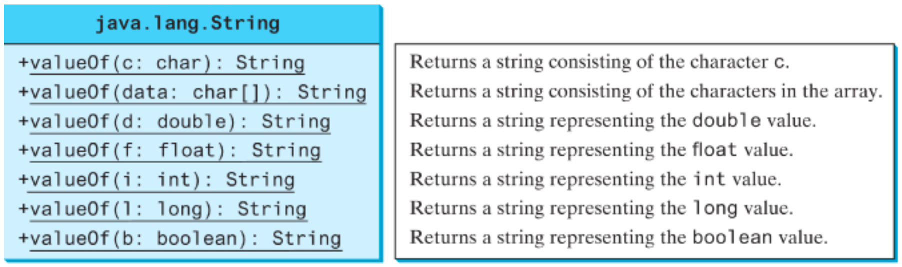
```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```


s1
s3
: String
Interned string object for "Welcome to Java"
s2
: String
A string object for "Welcome to Java"

```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, s1 and s3 refer to the same interned string—"Welcome to Java"—so s1 == s3 is true. However, s1 == s2 is false, because s1 and s2 are two different string objects, even though they have the same contents. Strings are not arrays, but a string can be converted into an array and vice versa. To convert a string into an array of characters, use the **toCharArray** method.

Another way of converting a number into a string is to use the overloaded static **valueOf** method. This method can also be used to convert a character or an array of characters into a string.



java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character c.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the double value.
+valueOf(f: float): String	Returns a string representing the float value.
+valueOf(i: int): String	Returns a string representing the int value.
+valueOf(l: long): String	Returns a string representing the long value.
+valueOf(b: boolean): String	Returns a string representing the boolean value.

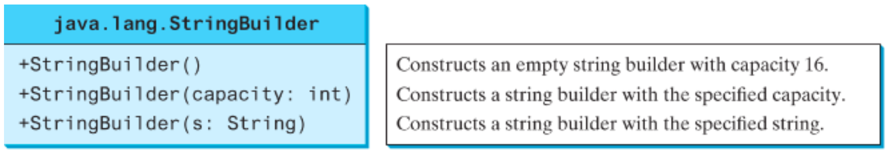
The String class contains the static format method to return a formatted string. This method is similar to the printf method except that the format method returns a formatted string, whereas the **printf** method displays a formatted string.

```
String s = String.format("%7.2f%6d%-4s", 45.556, 14, "AB");
System.out.println(s);
 --45.56----   14AB--
```

The StringBuilder and StringBuffer classes are similar to the String class except that the String class is immutable. In general, the StringBuilder and StringBuffer classes can be used wherever a string is used. StringBuilder and StringBuffer are more flexible than String. You can add, insert, or append new contents into StringBuilder and StringBuffer objects, whereas the value of a String object is fixed once the string is created.

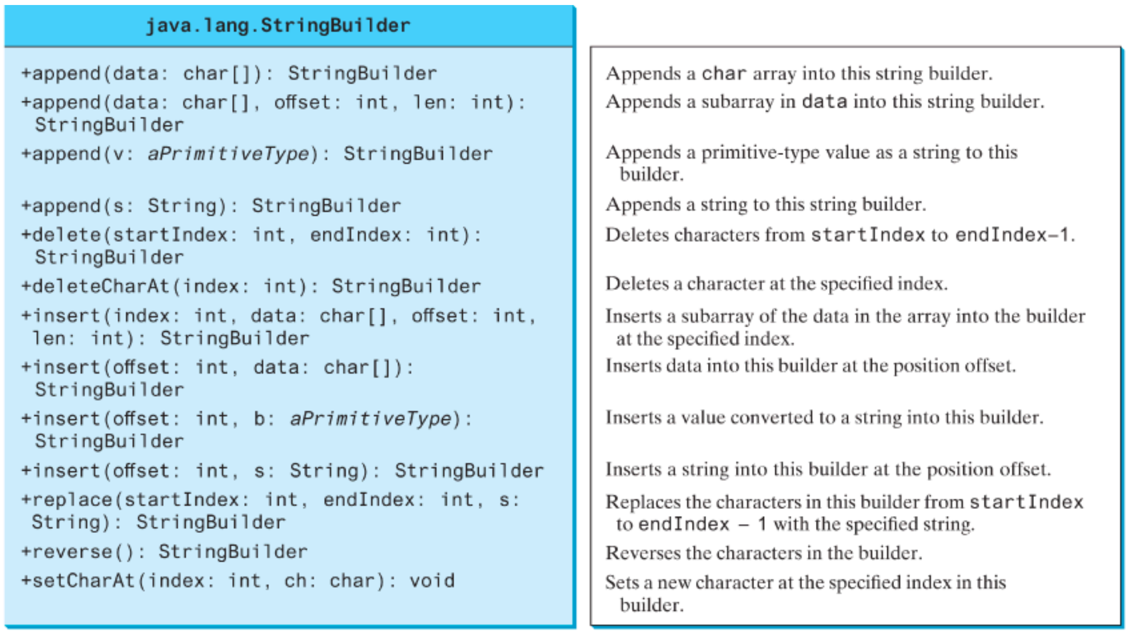
The StringBuilder class is similar to StringBuffer except that the methods for modifying the buffer in StringBuffer are synchronized, which means that only one task is allowed to execute the methods. Use StringBuffer if the class might be accessed by multiple tasks concurrently, because synchronization is needed in this case to prevent corruptions to StringBuffer. Using StringBuilder is more efficient if it is accessed by just a single task, because no synchronization is needed in this case. The constructors and methods in StringBuffer and StringBuilder are almost the same. This section covers StringBuilder. You can replace StringBuilder in all occurrences in this section by StringBuffer. The program can compile and run without any other changes.

The StringBuilder class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder or a string builder from a string using the constructors.



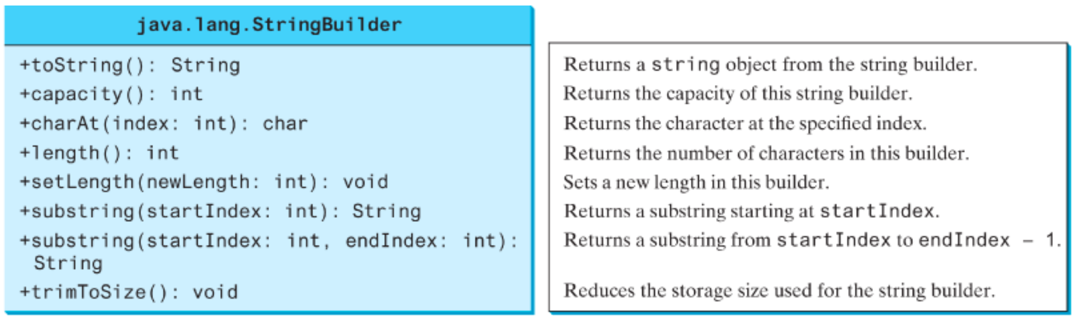
java.lang.StringBuilder	
+StringBuilder()	Constructs an empty string builder with capacity 16.
+StringBuilder(capacity: int)	Constructs a string builder with the specified capacity.
+StringBuilder(s: String)	Constructs a string builder with the specified string.

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed



java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a char array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in data into this string builder.
+append(v: *aPrimitiveType*): StringBuilder	Appends a primitive-type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from startIndex to endIndex-1.
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array into the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: *aPrimitiveType*): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from startIndex to endIndex – 1 with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

The StringBuilder class provides the additional methods for manipulating a string builder and obtaining its properties.



java.lang.StringBuilder	
+toString(): String	Returns a string object from the string builder.
+capacity(): int	Returns the capacity of this string builder.
+charAt(index: int): char	Returns the character at the specified index.
+length(): int	Returns the number of characters in this builder.
+setLength(newLength: int): void	Sets a new length in this builder.
+substring(startIndex: int): String	Returns a substring starting at startIndex.
+substring(startIndex: int, endIndex: int): String	Returns a substring from startIndex to endIndex – 1.
+trimToSize(): void	Reduces the storage size used for the string builder.

The length of the string builder is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is 2 * (the previous array size + 1).

You can use new StringBuilder(initialCapacity) to create a StringBuilder with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the trimToSize() method to reduce the capacity to the actual size.