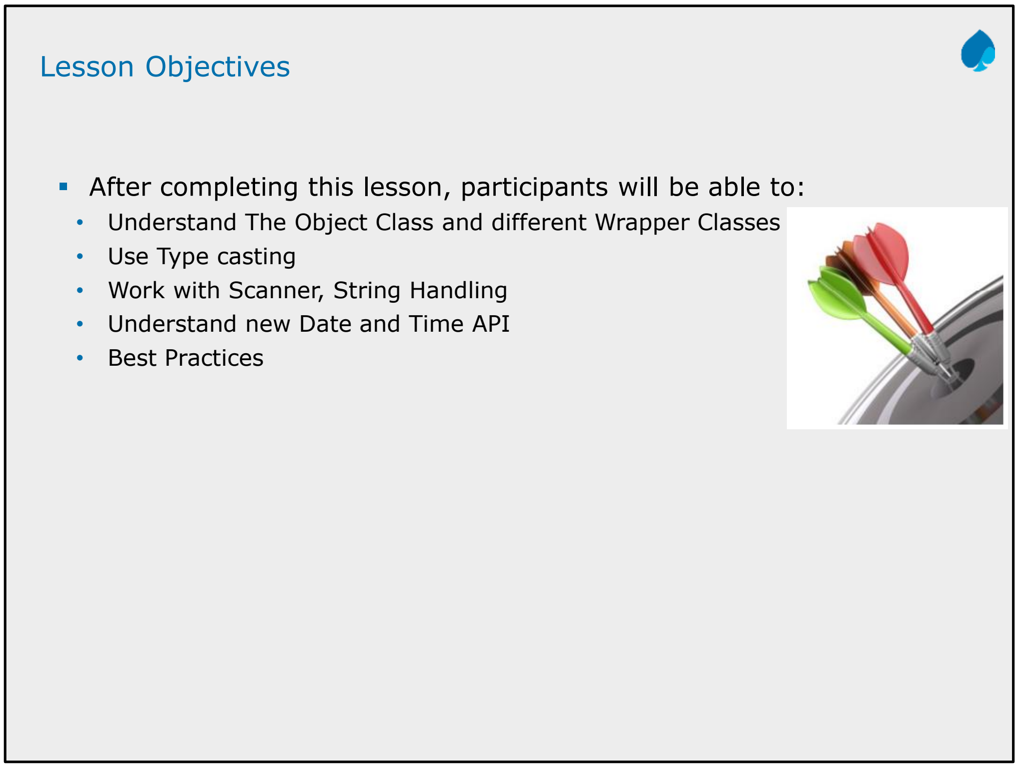
Core Java 8 and Development Tools

Lesson 05 : Exploring Basic Java Class Libraries

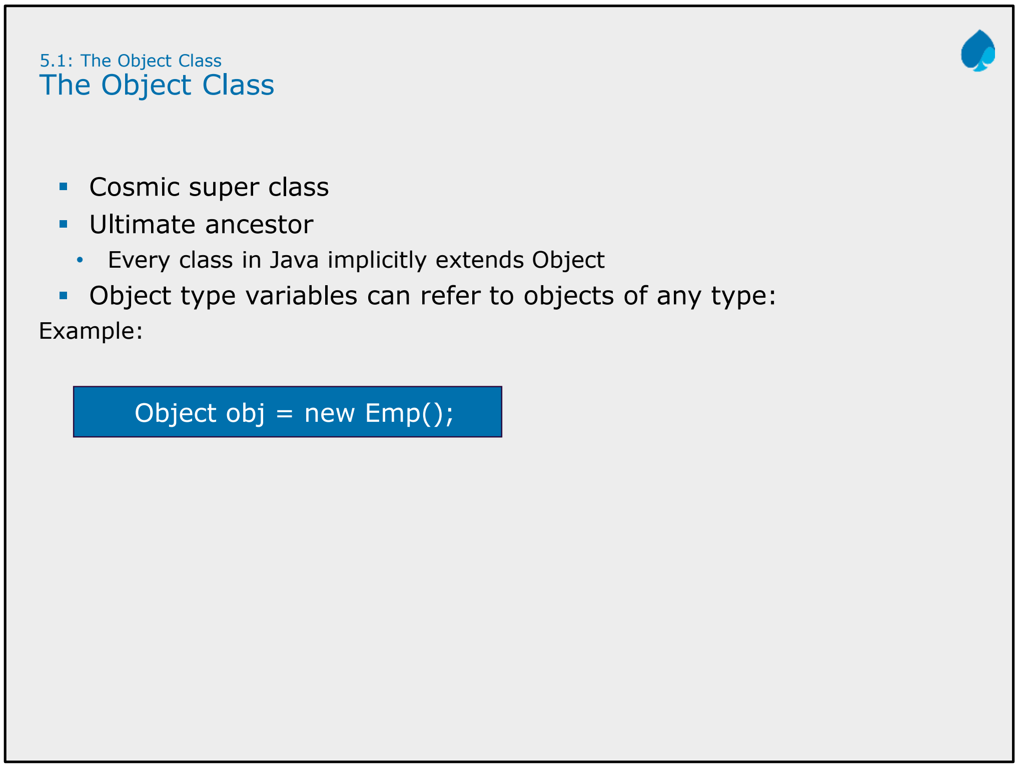
Capgemini



Lesson Objectives

- After completing this lesson, participants will be able to:
  - Understand The Object Class and different Wrapper Classes
  - Use Type casting
  - Work with Scanner, String Handling
  - Understand new Date and Time API
  - Best Practices

Lesson Objectives:
This lesson introduces to the fundamental Java API that is used in almost every type of Java applications.



5.1: The Object Class

The Object Class

- Cosmic super class
- Ultimate ancestor
  - Every class in Java implicitly extends Object
- Object type variables can refer to objects of any type:

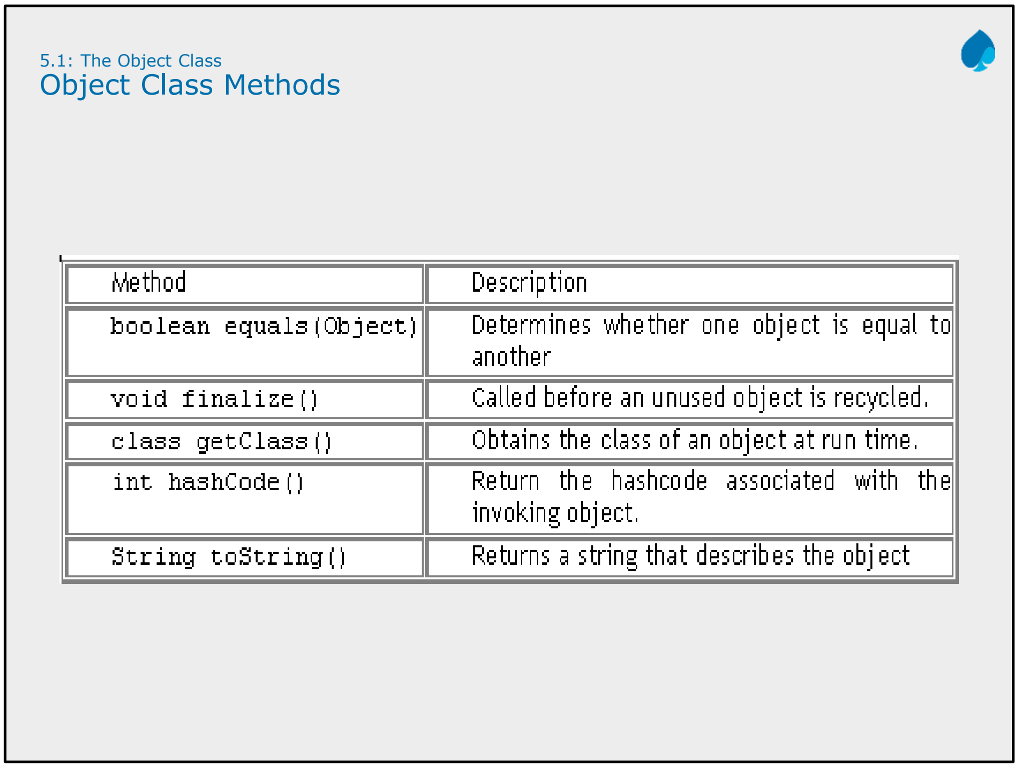
Example:

```
Object obj = new Emp();
```

The Object super class is a cosmic super class. Every class in Java extends Object class. It means a reference of object type can refer to an object of any other class. In addition, an array can be referenced by an object reference.

Example:

```
Object Obj1 = new Box(…….);
```

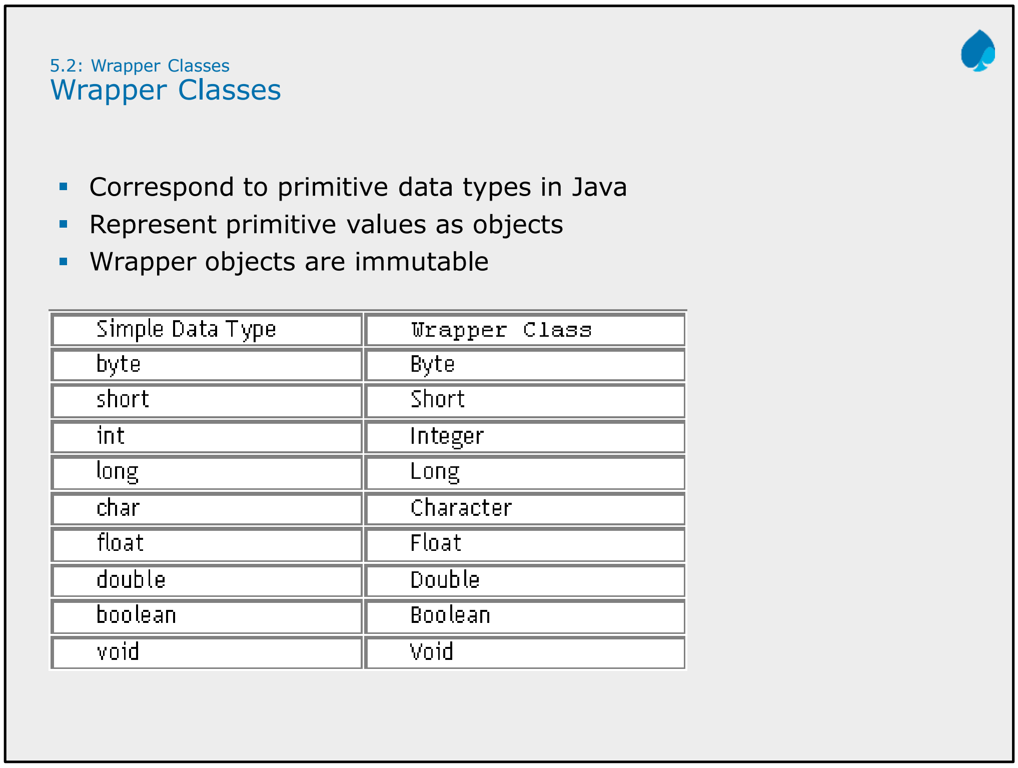


5.1: The Object Class

Object Class Methods

Method	Description
`boolean equals(Object)`	Determines whether one object is equal to another
`void finalize()`	Called before an unused object is recycled.
`class getClass()`	Obtains the class of an object at run time.
`int hashCode()`	Return the hashcode associated with the invoking object.
`String toString()`	Returns a string that describes the object

The table above shows some of the methods of the Object superclass. Please refer to Java docs for the entire list.



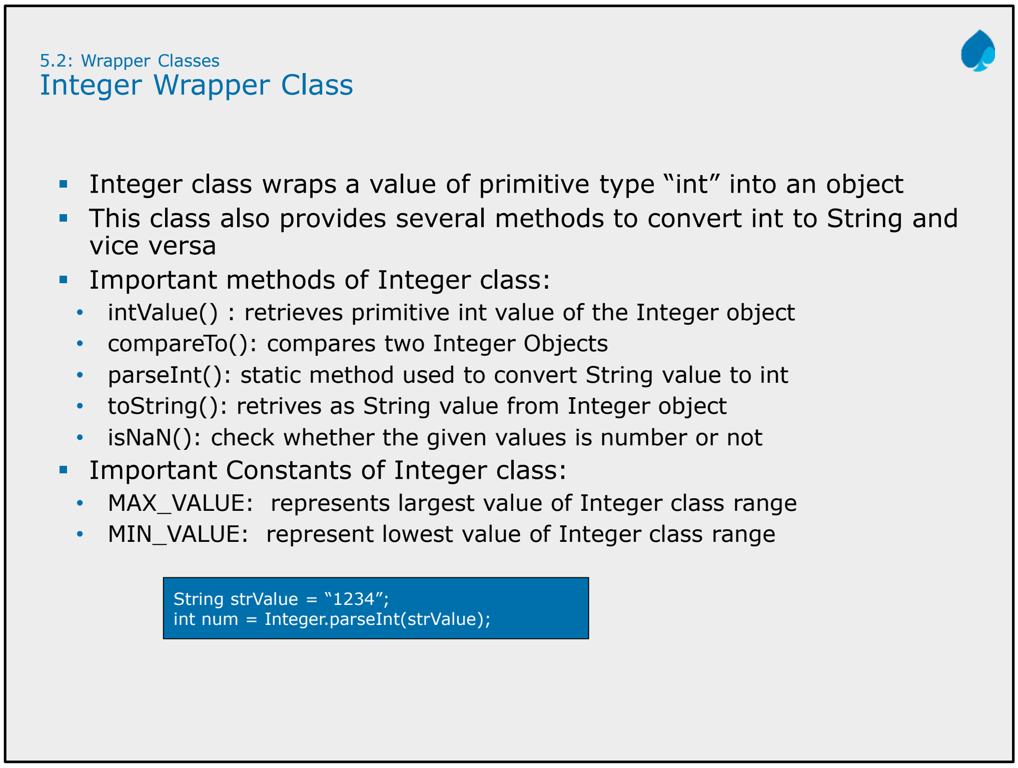
5.2: Wrapper Classes

Wrapper Classes

- Correspond to primitive data types in Java
- Represent primitive values as objects
- Wrapper objects are immutable

Simple Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
char	Character
float	Float
double	Double
boolean	Boolean
void	Void

Wrapper classes correspond to the primitive data types in the Java language. These classes represent the primitive values as objects. Wrapper objects are immutable. This means that once a wrapper object has a value assigned to it, that value cannot be changed.
Java uses simple or primitive data types, such as int, char and Boolean etc. These data types are not part of the object hierarchy. They are passed by value to methods and cannot be directly passed by reference. However, at times there is a need to create an object representation of these simple data types. Java provides classes that correspond to each of these simple types. These classes encapsulate, or wrap, the simple data type within a class. Thus, they are commonly referred to as wrapper classes. The abstract class Number defines a superclass that is implemented by all numeric wrapper classes.



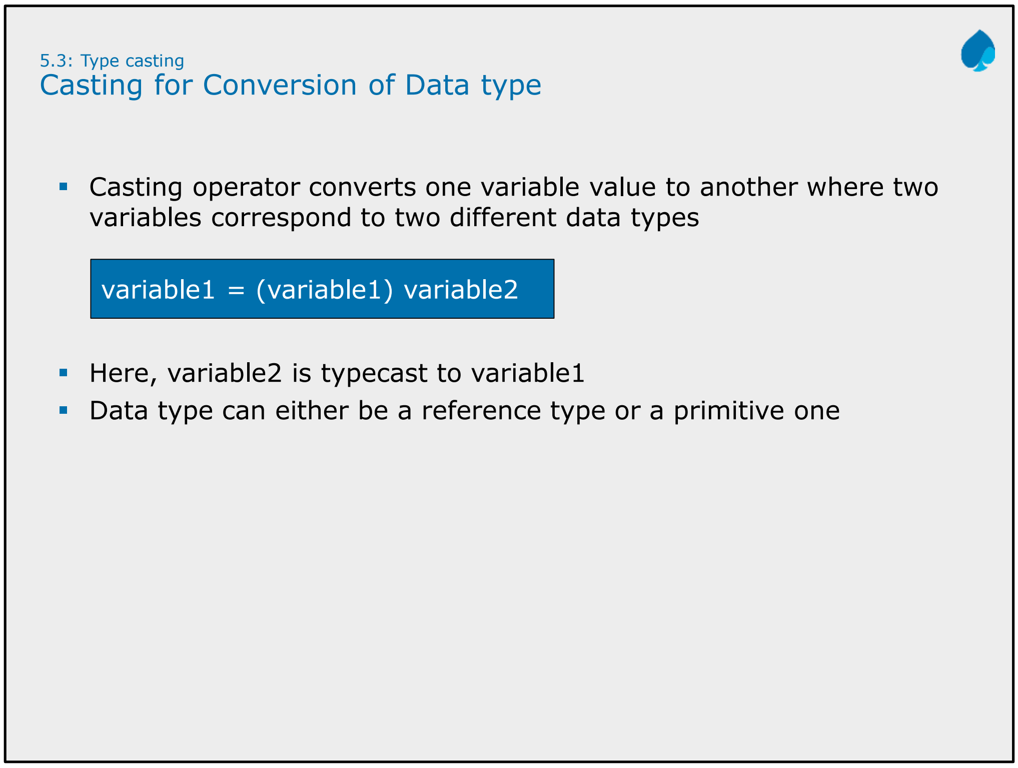
5.2: Wrapper Classes

Integer Wrapper Class

- Integer class wraps a value of primitive type “int” into an object
- This class also provides several methods to convert int to String and vice versa
- Important methods of Integer class:
  - intValue() : retrieves primitive int value of the Integer object
  - compareTo(): compares two Integer Objects
  - parseInt(): static method used to convert String value to int
  - toString(): retrives as String value from Integer object
  - isNaN(): check whether the given values is number or not
- Important Constants of Integer class:
  - MAX_VALUE: represents largest value of Integer class range
  - MIN_VALUE: represent lowest value of Integer class range

```
String strValue = “1234”;
int num = Integer.parseInt(strValue);
```

Above slide represents methods and constants provided by Integer wrapper class. Other wrapper classes too have similar kind of method and constant structure. The method names varies according to the wrapper class name. For example, intValue() method is anonymous to longValue() of Long wrapper class.



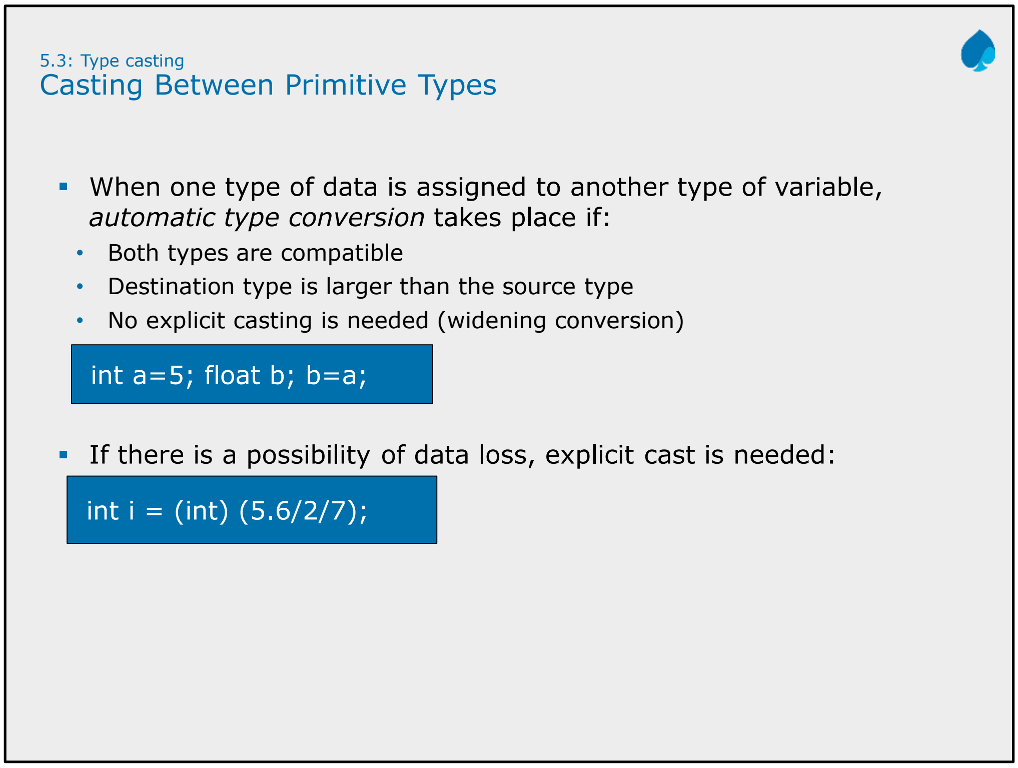
5.3: Type casting

Casting for Conversion of Data type

- Casting operator converts one variable value to another where two variables correspond to two different data types

```
variable1 = (variable1) variable2
```

- Here, variable2 is typecast to variable1
- Data type can either be a reference type or a primitive one



5.3: Type casting

Casting Between Primitive Types

- When one type of data is assigned to another type of variable, *automatic type conversion* takes place if:
  - Both types are compatible
  - Destination type is larger than the source type
  - No explicit casting is needed (widening conversion)

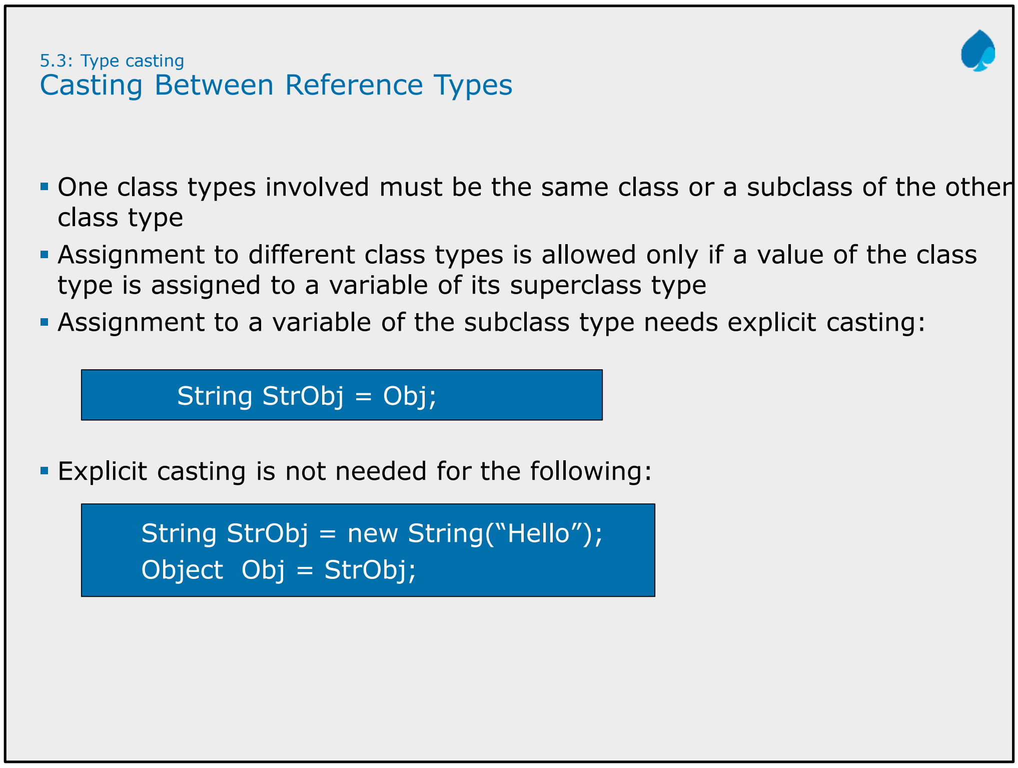
```
int a=5; float b; b=a;
```

- If there is a possibility of data loss, explicit cast is needed:

```
int i = (int) (5.6/2/7);
```

When one type of data is assigned to another type of variable, an automatic type conversion will take place if the following two conditions are met:
The two types are compatible.
The destination type is larger than the source type.
In this case, a widening conversion takes place. For example, the int type is always large enough to hold all valid byte values, so no explicit cast statement is required.
To widen conversions, numeric types, including integer and floating-point types, are compatible with each other. However, numeric types are not compatible with char or boolean. Also, char and boolean are not compatible with each other.
As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type byte, short, or long.
Casting Incompatible Types
Automatic type conversions may not fulfill all needs though. For example, assigning an int value to a byte variable. This conversion will not be performed automatically, because a byte is smaller than an int. This is called a narrowing conversion, since you are explicitly making the value narrower so that it will fit into the target type. This is done using a cast - an explicit type conversion. It has this general form: (target-type) value
Here, target-type specifies the desired type to convert the specified value to. For example, the following fragment casts an int to a byte:

```
int i = 125;  byte b;
b = (byte) i;
```



5.3: Type casting

Casting Between Reference Types

- One class types involved must be the same class or a subclass of the other class type
- Assignment to different class types is allowed only if a value of the class type is assigned to a variable of its superclass type
- Assignment to a variable of the subclass type needs explicit casting:

```
String StrObj = Obj;
```

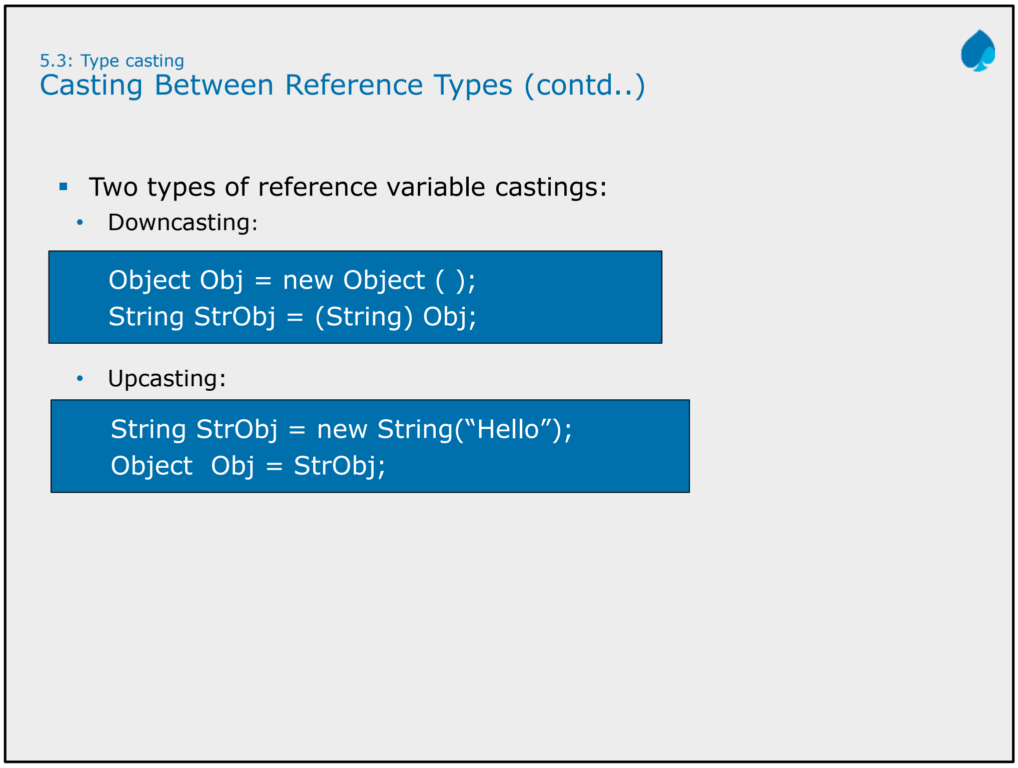
- Explicit casting is not needed for the following:

```
String StrObj = new String(“Hello”);
Object  Obj = StrObj;
```

The first rule of casting between reference types is that one of the class types involved must be the same class as, or a subclass of, the other class type. Assignment to different class type is allowed only if a value of the class type is assigned to a variable of its superclass type. Assignment to a variable of the subclass type needs explicit casting.

```
Object Obj = new Object ( );
String StrObj = Obj;
/*    The second statement of the above code is not a legitimate one, because
class String is a subclass of class object.  So, an explicit casting is needed. This
is called as downcasting
*/
String StrObj = (String) Obj;
//The reverse statement will not require casting. It is upcasting
String StrObj = new String(“Hello”);
Object  Obj = StrObj;
```

Rule number one in casting is that you cannot cast a primitive type to a reference type, nor can you cast the other way around. If a casting violation is detected at runtime, the exception ClassCastException is thrown.



5.3: Type casting

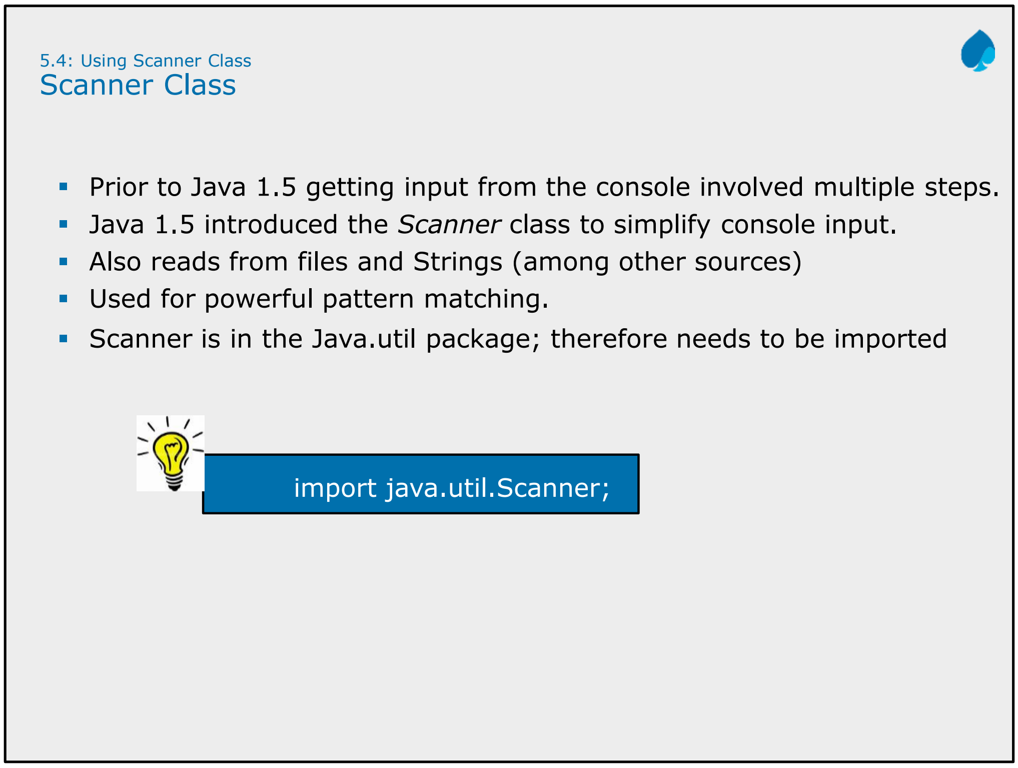
Casting Between Reference Types (contd..)

- Two types of reference variable castings:
  - Downcasting:

```
Object Obj = new Object ( );
String StrObj = (String) Obj;
```

  - Upcasting:

```
String StrObj = new String(“Hello”);
Object  Obj = StrObj;
```

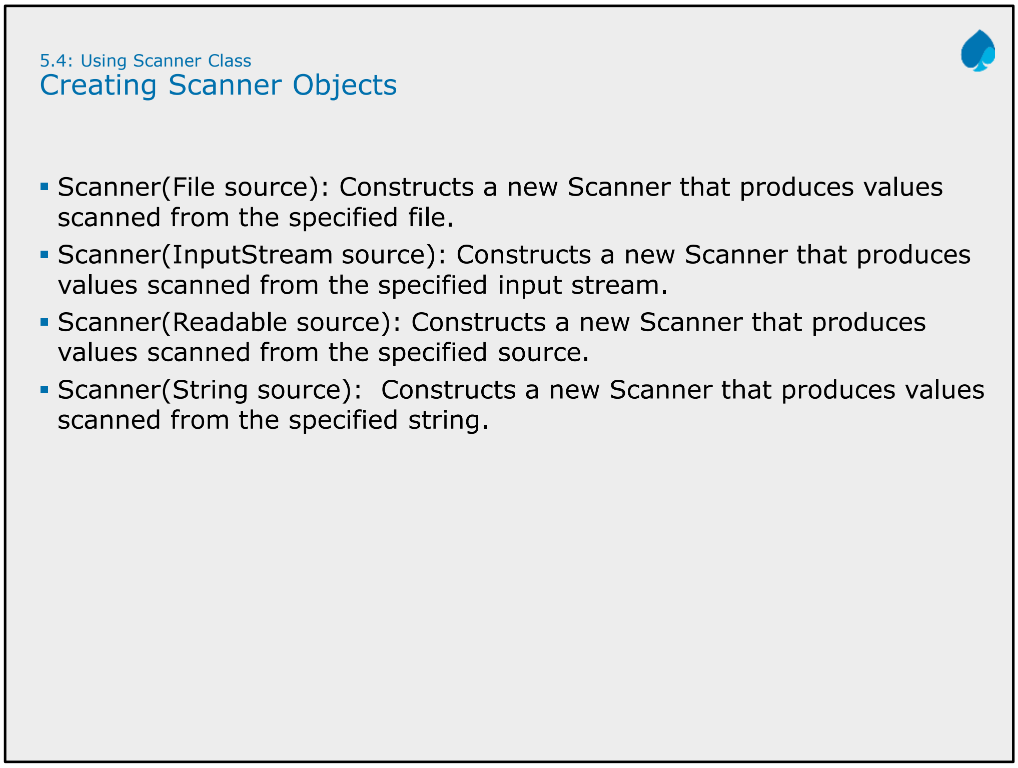


5.4: Using Scanner Class

Scanner Class

- Prior to Java 1.5 getting input from the console involved multiple steps.
- Java 1.5 introduced the *Scanner* class to simplify console input.
- Also reads from files and Strings (among other sources)
- Used for powerful pattern matching.
- Scanner is in the Java.util package; therefore needs to be imported

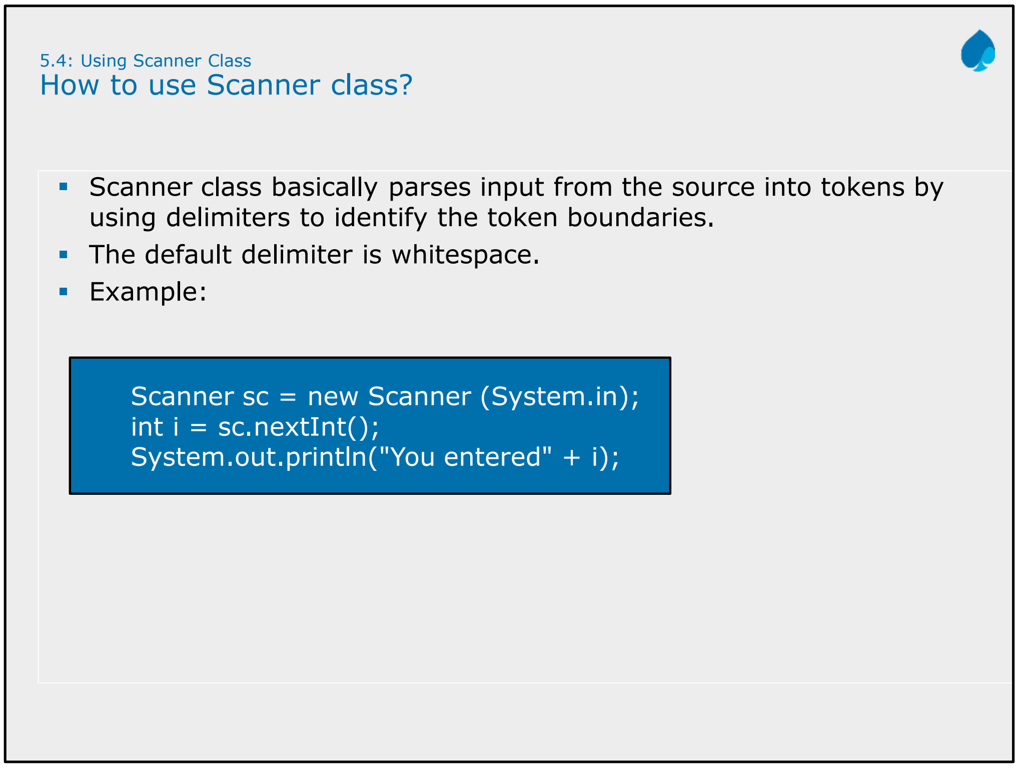
```
import java.util.Scanner;
```



5.4: Using Scanner Class

Creating Scanner Objects

- Scanner(File source): Constructs a new Scanner that produces values scanned from the specified file.
- Scanner(InputStream source): Constructs a new Scanner that produces values scanned from the specified input stream.
- Scanner(Readable source): Constructs a new Scanner that produces values scanned from the specified source.
- Scanner(String source): Constructs a new Scanner that produces values scanned from the specified string.



5.4: Using Scanner Class

How to use Scanner class?

- Scanner class basically parses input from the source into tokens by using delimiters to identify the token boundaries.
- The default delimiter is whitespace.
- Example:

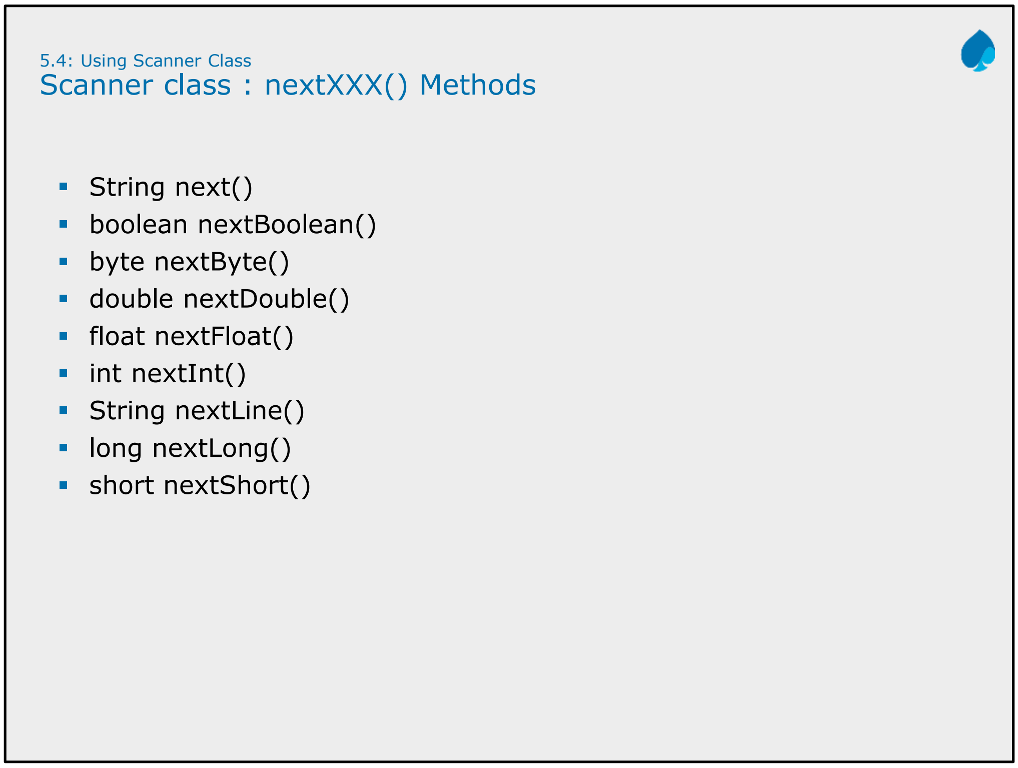
```
Scanner sc = new Scanner (System.in);
int i = sc.nextInt();
System.out.println("You entered" + i);
```

Example 1:

```
import java.util.Scanner;
public class ParseString
{
    public static void main(String[] args)
    {
        private static Scanner scanner = new
                                     Scanner("1 2 3 4 5 6 7 8");

        while (scanner.hasNextInt()) {
            int num = scanner.nextInt();

            if (num % 2 == 0)
                System.out.print(num);
        }
    }
}
```

Output:

```
2
4
6
8
```



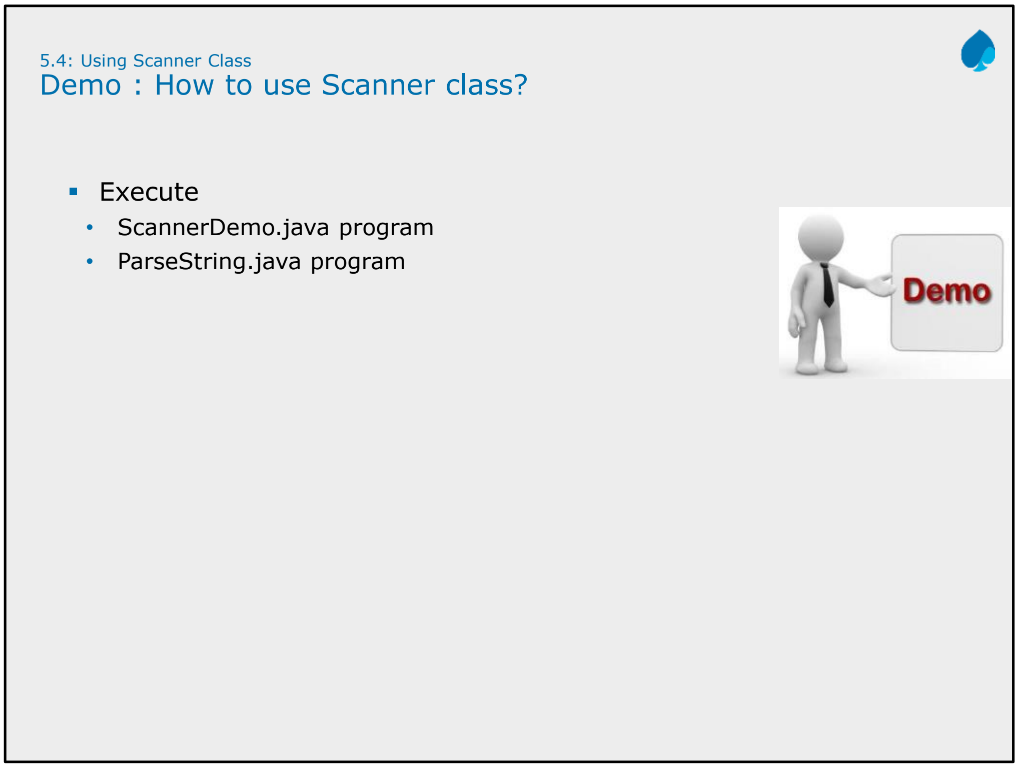
5.4: Using Scanner Class

Scanner class : nextXXX() Methods

- String next()
- boolean nextBoolean()
- byte nextByte()
- double nextDouble()
- float nextFloat()
- int nextInt()
- String nextLine()
- long nextLong()
- short nextShort()

String next(): Finds and returns the next complete token from this scanner.
boolean nextBoolean(): Scans the next token of the input into a boolean value and returns that value.
byte nextByte(): Scans the next token of the input as a byte.
double nextDouble(): Scans the next token of the input as a double.
float nextFloat(): Scans the next token of the input as a float.
int nextInt(): Scans the next token of the input as an int.
String nextLine(): Advances this scanner past the current line and returns the input that was skipped.
long nextLong(): Scans the next token of the input as a long.
short nextShort(): Scans the next token of the input as a short.

InputMismatchException: This exception can be thrown if you try to get the next token using a next method that does not match the type of the token



5.4: Using Scanner Class

Demo : How to use Scanner class?

- Execute
  - ScannerDemo.java program
  - ParseString.java program

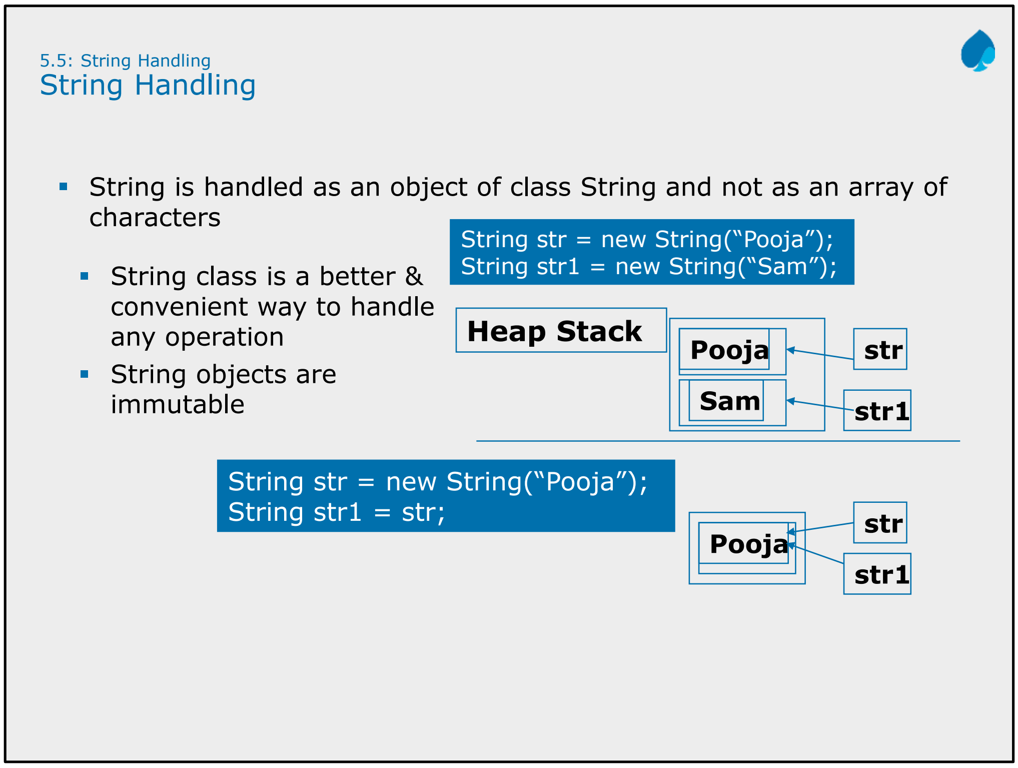
When you create an instance of the Scanner class, the default delimiter is whitespace. The Scanner class provides the useDelimiter method for specifying the Delimiter.

```
import java.util.Scanner;
public class ParseString
{
    public static void main(String[] args)
    {
       Scanner scanner =  new Scanner("1, 2, 3, 4, 5, 6,
7,8").useDelimiter(", ");

        while (scanner.hasNextInt()) {
            int num = scanner.nextInt();

            if (num % 2 == 0)
                System.out.println(num);
        }
    }
}
```

Demo

- Execute the Elapsed.java program

```
class Elapsed {
  public static void main(String args[]) throws IOException {
    long start, end;
    int i = 0, sum = 0;
    String str = null;
     System.out.println("Timing a for loop from 0 to 1,000,000");
     // time a for loop from 0 to 1,000,000
    start = System.currentTimeMillis(); // get starting time
    for(int j=0; j < 1000000; j++) ;
   end = System.currentTimeMillis(); // get ending time
   System.out.println("Elapsed time: " + (end-start));
   // Demo to read from the system input and write to standard output.
   BufferedReader br = new
                BufferedReader(new InputStreamReader(System.in));
    do {
         System.out.println("Enter 0 to quit");
         str = br.readLine();
          i = Integer.parseInt(str);
          if ( i == 0 )   System.exit(0); // normal exit
          sum += i;
          System.out.println("Current sum is: " + sum);
     } while(i != 0);
}}
```



5.5: String Handling

String Handling

- String is handled as an object of class String and not as an array of characters
  - String class is a better & convenient way to handle any operation
  - String objects are immutable

```
String str = new String("Pooja");
String str1 = new String("Sam");
```

Heap Stack
Pooja
Sam
str
str1

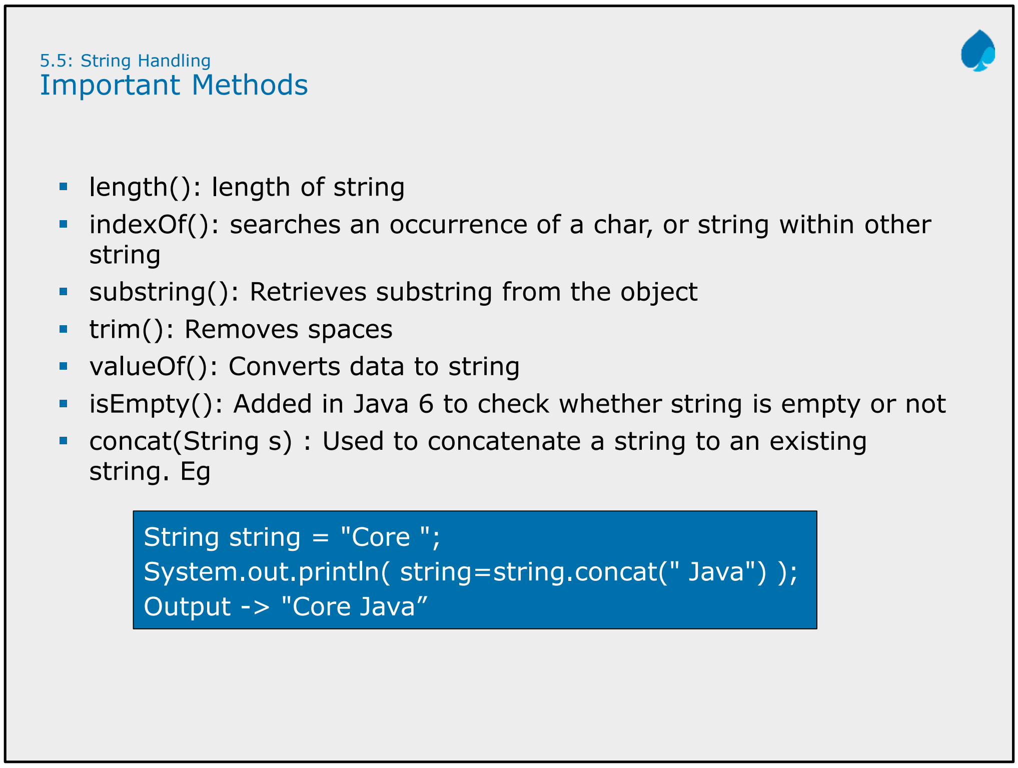
```
String str = new String("Pooja");
String str1 = str;
```

Pooja
str
str1

String is not an array of characters but it is actually a class and is part of core API. We can use the class String as a usual data-type. It can store up to 2 billion characters.
Note: A String in java is not equivalent to character array.
Strings are built-in objects & thus have a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a sub string, concatenate two strings etc. In addition, String objects can be constructed in number of ways.

String objects are immutable objects. That is, once you create a String object you cannot change the characters, which are part of String. This seems to be a major restriction, but that is not the case. Every time you perform some modification operation on the object, a new String object is created that contains the modifications. The original string is left unchanged. Hence, the number of operations performed on one particular string creates those many string objects.

For those cases in which modifiable string is desired, there is a companion class to String called StringBuffer, whose objects contain strings that can be modified after they are created.



5.5: String Handling

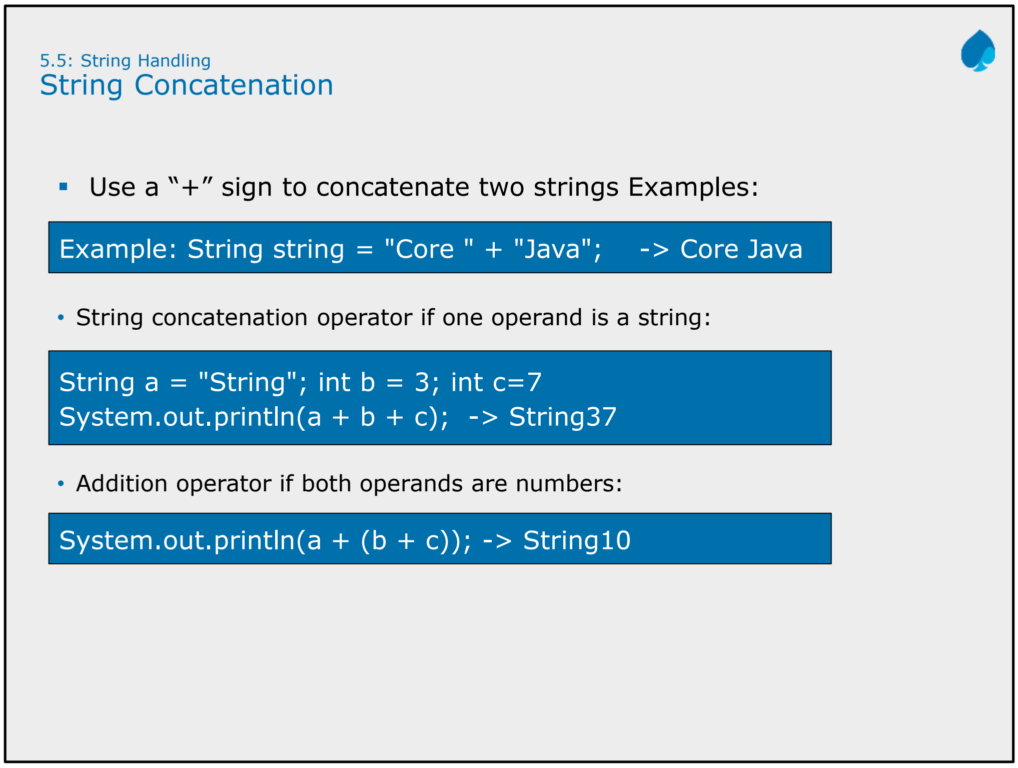
Important Methods

- length(): length of string
- indexOf(): searches an occurrence of a char, or string within other string
- substring(): Retrieves substring from the object
- trim(): Removes spaces
- valueOf(): Converts data to string
- isEmpty(): Added in Java 6 to check whether string is empty or not
- concat(String s) : Used to concatenate a string to an existing string. Eg

```
String string = "Core ";
System.out.println( string=string.concat(" Java") );
Output -> "Core Java”
```

String.isEmpty() method added in Java 6 to check whether the given string is empty or not. Code prior to JDK 6 is as shown below:

```
//code prior to JDK 6

public boolean checkStringForEmpty(String str) {
    If(str.equals(“”)) {            //str.length==0
        return true;
    else
        return false;
}


//now with JDK 6 enhancement

public boolean checkStringForEmpty(String str) {
    If(str.isEmpty()) {             //much faster than the previous code
        return true;
    else
        return false;
}
```



5.5: String Handling

String Concatenation

- Use a "+" sign to concatenate two strings Examples:

```
Example: String string = "Core " + "Java";    -> Core Java
```

- String concatenation operator if one operand is a string:

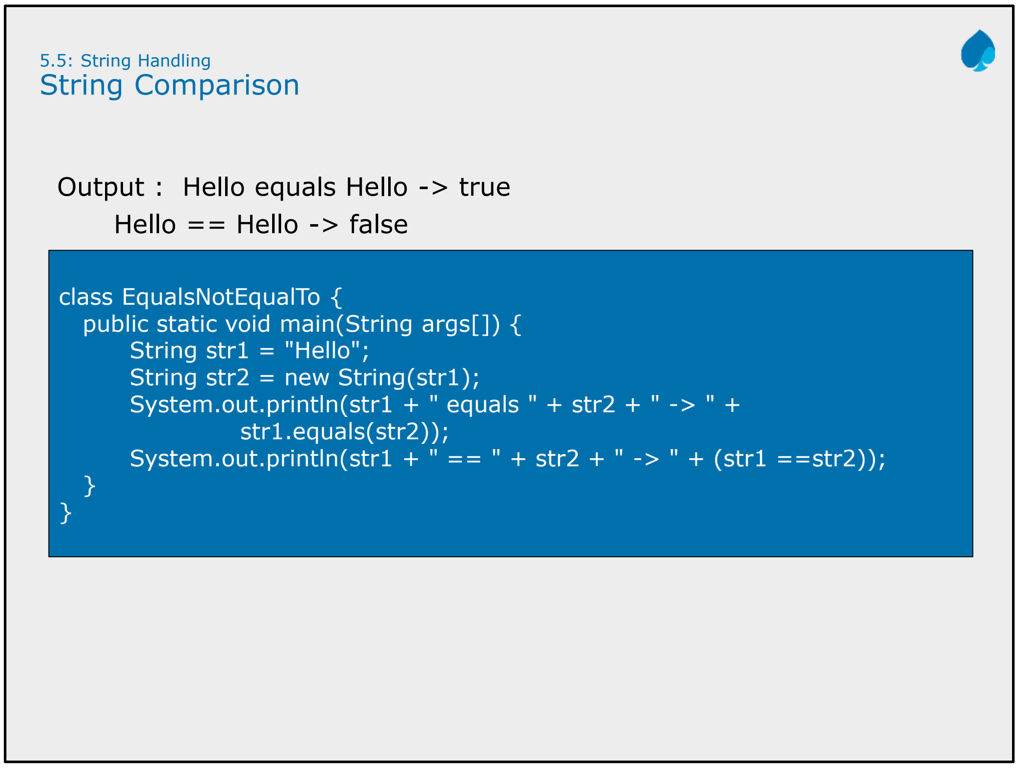
```
String a = "String"; int b = 3; int c=7
System.out.println(a + b + c);  -> String37
```

- Addition operator if both operands are numbers:

```
System.out.println(a + (b + c)); -> String10
```

The concat() method seen in previous page allows one string to be concatenated to another. But Java also supports string concatenation with the "+" operator.
In general, Java does not support operator overloading. The exception to this rule is the + operator, which concatenates two strings, and produces a new string object as a result.

```
class SimpleString  {
    public static void main(String args[])  {
        // Simple String Operations
        char c[] = {'J', 'a', 'v', 'a'};
        String s1 = new String(c); // String constructor using
        String s2 = new String(s1);
        // String constructor using string as arg.
        System.out.println(s1);
        System.out.println(s2);
        System.out.println("Length of String s2 : " + s2.length());
        System.out.println("Index of v : " + s2.indexOf('v'));
        System.out.println("s2 in uppercase : " + s2.toUpperCase());
        System.out.println("Character at position 2 is  : " +  s2.charAt(1));
        // Using concatenation to prevent long lines.
         String longStr = "This could have been " +
                     "a very long line that would have " +
                     "wrapped around.  But string concatenation " +
                     "prevents this.";
          System.out.println(longStr);
    } }
```



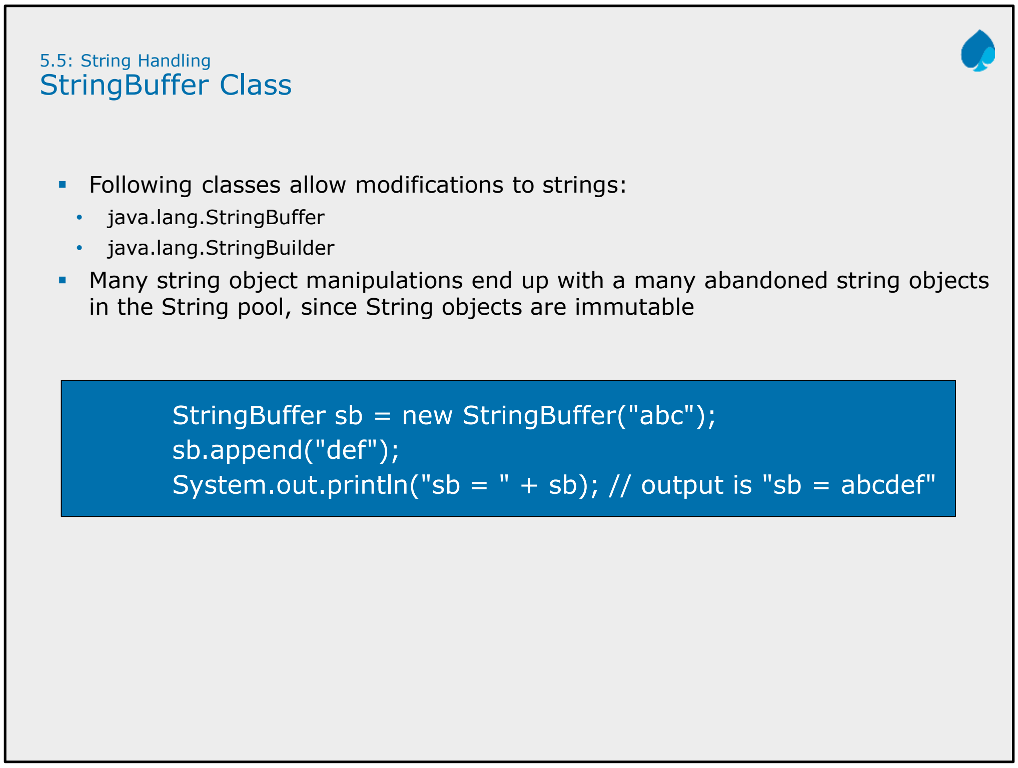
5.5: String Handling

String Comparison

```
Output :  Hello equals Hello -> true
     Hello == Hello -> false
```

```
class EqualsNotEqualTo {
  public static void main(String args[]) {
      String str1 = "Hello";
      String str2 = new String(str1);
      System.out.println(str1 + " equals " + str2 + " -> " +
                 str1.equals(str2));
      System.out.println(str1 + " == " + str2 + " -> " + (str1 ==str2));
  }
}
```

The String class includes various methods that compare strings or substrings within each string. The most popularly used two ways to compare the strings is either using = = operator or by using the equals method.
The equals() method compares the characters inside a String object. The = = operator compare two object references to see whether they refer to the same instance. The program above shows the difference between the two.



5.5: String Handling

StringBuffer Class

- Following classes allow modifications to strings:
  - java.lang.StringBuffer
  - java.lang.StringBuilder
- Many string object manipulations end up with a many abandoned string objects in the String pool, since String objects are immutable

```
StringBuffer sb = new StringBuffer("abc");
sb.append("def");
System.out.println("sb = " + sb); // output is "sb = abcdef"
```

Let us understand StringBuilder with an example.

```
String x = "abc";
x.concat("def");
System.out.println("x = " + x); // output is "x = abc"
```

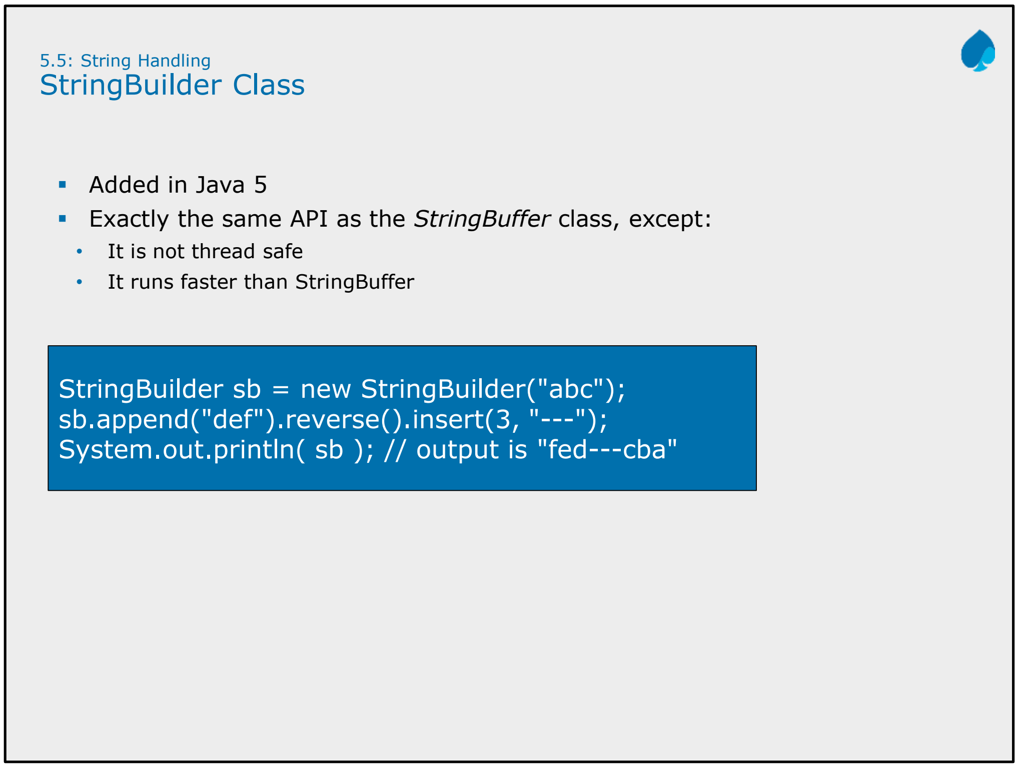
Because no new assignment was made, the new String object created with the concat() method was abandoned instantly. We also saw examples like this:

```
String x = "abc";
x = x.concat("def");
System.out.println("x = " + x); // output is "x = abcdef"
```

We got a nice new String out of the deal, but the downside is that the old String "abc" has been lost in the String pool, thus wasting memory. If we were using a StringBuffer instead of a String, the code would look like this:

```
StringBuffer sb = new StringBuffer("abc");
        sb.append("def");
       System.out.println("sb = " + sb); // output is "sb = abcdef“
```

Note: Refer Javadocs to know more about other methods of StringBuilder.



5.5: String Handling

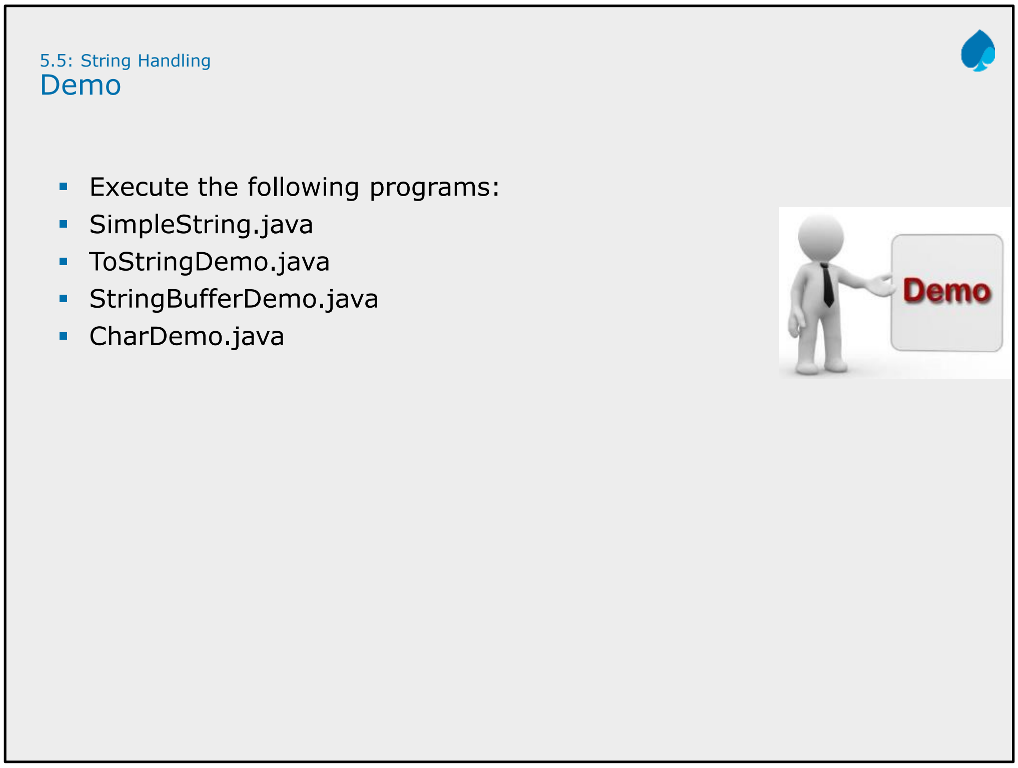
StringBuilder Class

- Added in Java 5
- Exactly the same API as the *StringBuffer* class, except:
  - It is not thread safe
  - It runs faster than StringBuffer

```
StringBuilder sb = new StringBuilder("abc");
sb.append("def").reverse().insert(3, "---");
System.out.println( sb ); // output is "fed---cba"
```

The StringBuilder class was added in Java 5. It has exactly the same API as the StringBuffer class, except StringBuilder is not thread safe. In other words, its methods are not synchronized. Sun recommends that you use StringBuilder instead of StringBuffer whenever possible because StringBuilder will run faster (and perhaps jump higher). So, apart from
Synchronization, anything we say about StringBuilder's methods holds true for StringBuffer's methods, and vice versa.

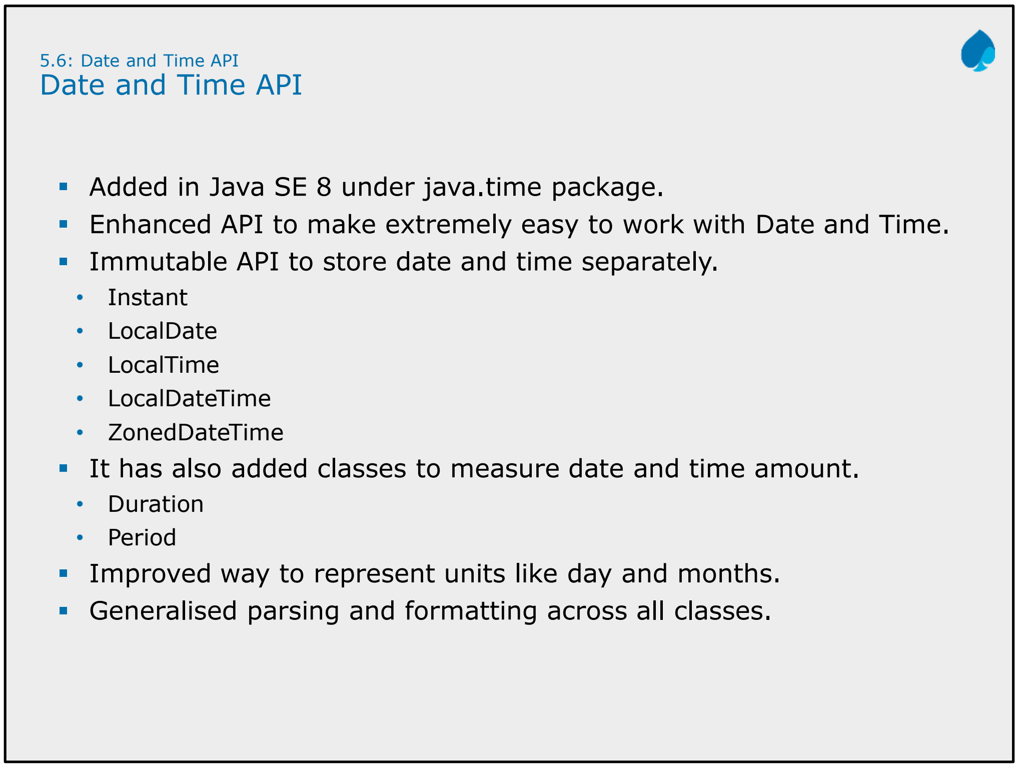
Note: Refer Javadocs to know more about methods of StringBuilder.



5.5: String Handling

Demo

- Execute the following programs:
- SimpleString.java
- ToStringDemo.java
- StringBufferDemo.java
- CharDemo.java



5.6: Date and Time API

Date and Time API

- Added in Java SE 8 under java.time package.
- Enhanced API to make extremely easy to work with Date and Time.
- Immutable API to store date and time separately.
  - Instant
  - LocalDate
  - LocalTime
  - LocalDateTime
  - ZonedDateTime
- It has also added classes to measure date and time amount.
  - Duration
  - Period
- Improved way to represent units like day and months.
- Generalised parsing and formatting across all classes.

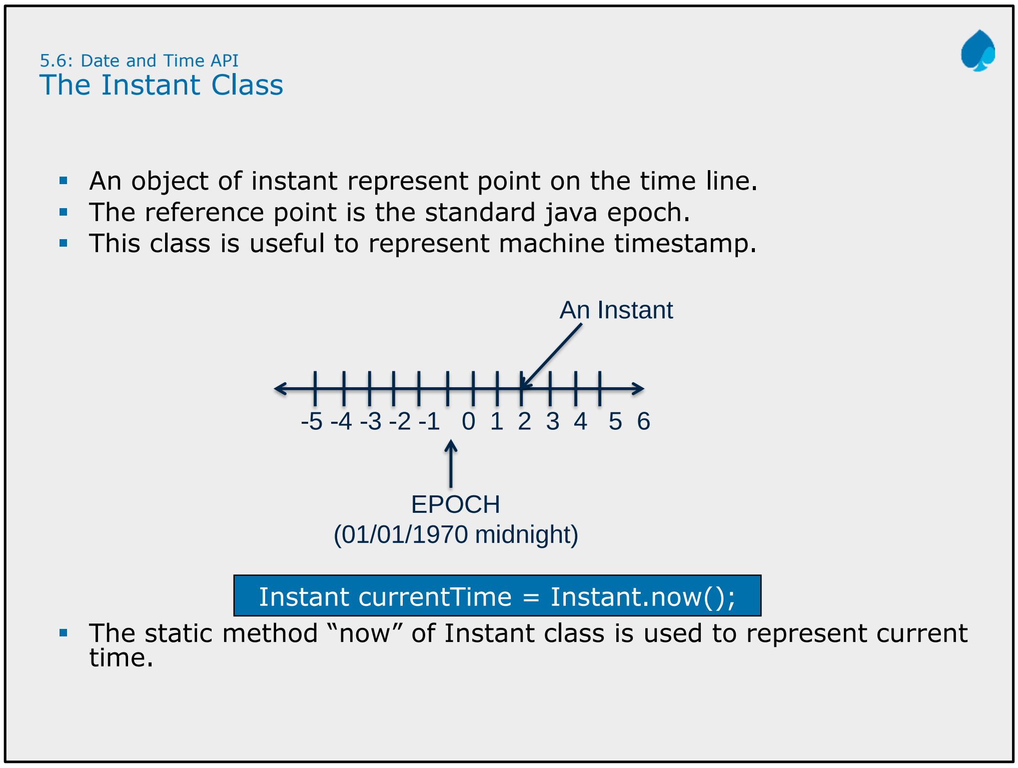
A long-standing bugbear of Java developers has been the inadequate support for the date and time. In order to address problems in legacy API (Date and Calender) and provide better support in the JDK core, a new date and time API (JSR 310) has been designed for Java SE 8 under java.time package.

LocalDate and LocalTime represents date and time respectively. The combination of date and time is represented by LocalDateTime. If a time zone is important, ZonedDateTime class is handy.

Many times developers need to measure amount of time between to date instants. Duration class used to measure amount of time including nanosecond precision. Period class is used to measure in terms of days, months or years.

This API has included two Enums DayOfWeek and Month to represent day and month constant respectively.

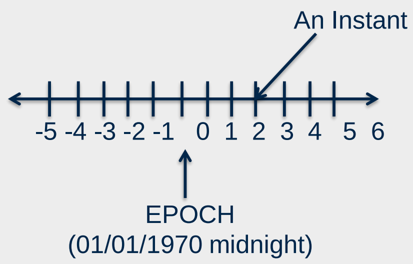
DateTimeFormatter class is used to format the date and times. Also almost all classes now include parse and format method to support parsing and formatting of date and time.



5.6: Date and Time API

The Instant Class

- An object of instant represent point on the time line.
- The reference point is the standard java epoch.
- This class is useful to represent machine timestamp.

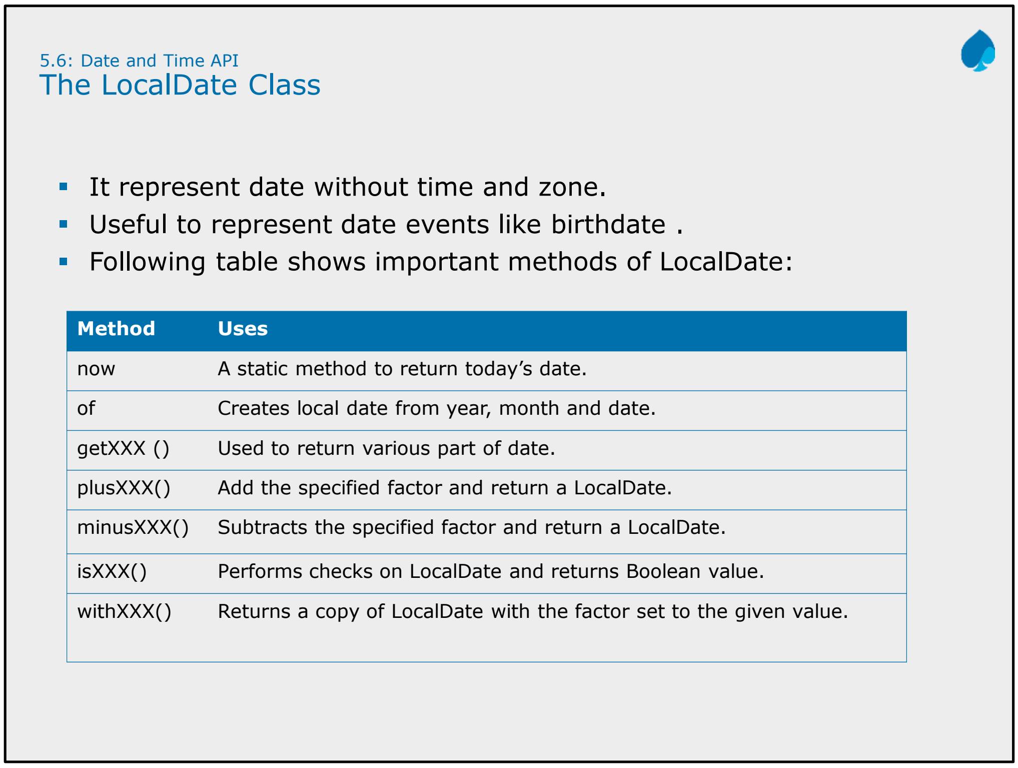


```
Instant currentTime = Instant.now();
```

- The static method "now" of Instant class is used to represent current time.

Instant class is useful for generating a time stamp to represent machine time. A value returned from the Instant class counts time beginning from the first second of January 1, 1970. This value is known as EPOCH.

An epoch is an instant on a timeline that is used as a reference point (or the origin) to measure other instants. As shown in the above figure, an Instant at epoch is represent by zero. Instants after epoch are positive numbers where as instants before epoch are negative.



5.6: Date and Time API

The LocalDate Class

- It represent date without time and zone.
- Useful to represent date events like birthdate .
- Following table shows important methods of LocalDate:

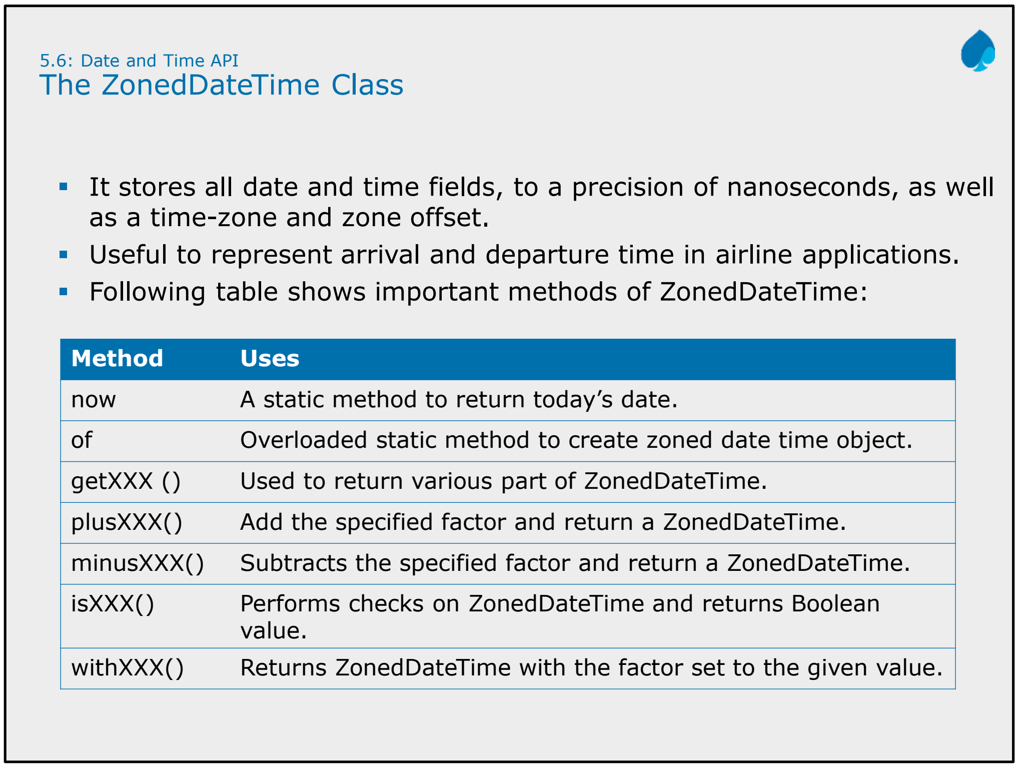
Method	Uses
now	A static method to return today's date.
of	Creates local date from year, month and date.
getXXX ()	Used to return various part of date.
plusXXX()	Add the specified factor and return a LocalDate.
minusXXX()	Subtracts the specified factor and return a LocalDate.
isXXX()	Performs checks on LocalDate and returns Boolean value.
withXXX()	Returns a copy of LocalDate with the factor set to the given value.

A LocalDate represents a year-month-day in the ISO calendar and is useful for representing a date without a time.

```java
LocalDate now = LocalDate.now();
LocalDate independence = LocalDate.of(1947, Month.AUGUST, 15);
System.out.println("Independence:"+ independence);
System.out.println("Today:"+now);
System.out.println("Tomorrow:"+ now.plusDays(1));
System.out.println("Last Month:"+ now.minusMonths(1));
System.out.println("Is leap?:"+ now.isLeapYear());
System.out.println("Move to 30th day of month:"+ now.withDayOfMonth(30));
```

The other two classes LocalTime and LocalDateTime as name reflects are used to store time and date with time respectively.

Most of the methods of LocalDateTime are analogous to LocalDate with few additional methods like plusHours(), plusMinutes() etc.



5.6: Date and Time API

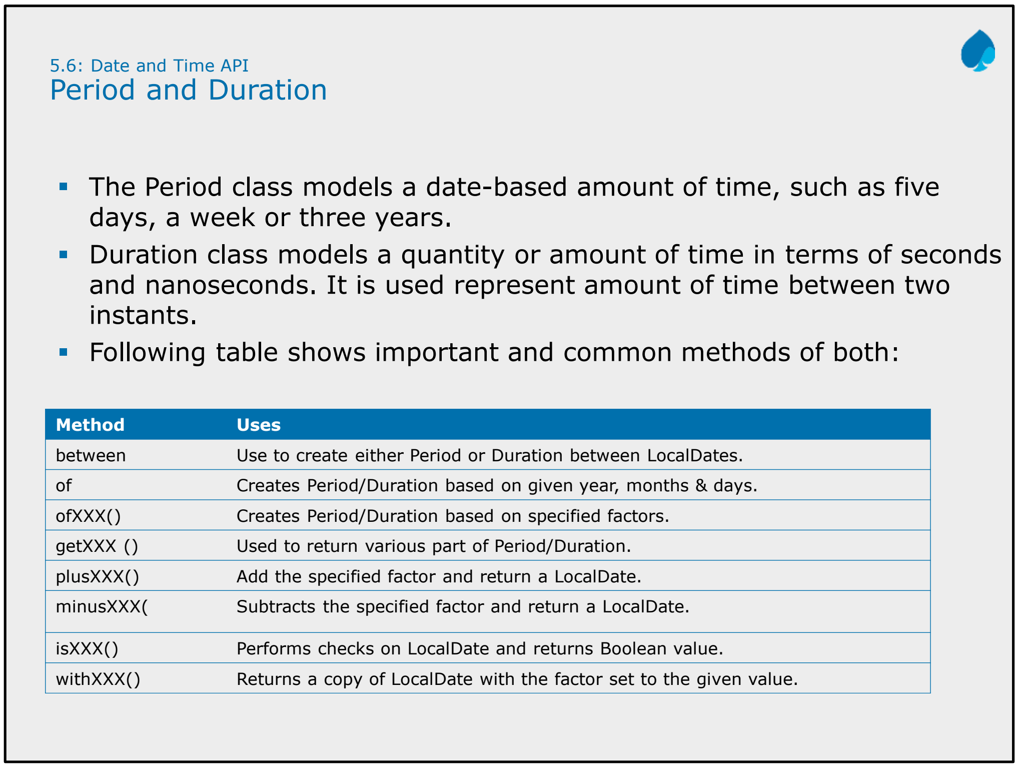
The ZonedDateTime Class

- It stores all date and time fields, to a precision of nanoseconds, as well as a time-zone and zone offset.
- Useful to represent arrival and departure time in airline applications.
- Following table shows important methods of ZonedDateTime:

Method	Uses
now	A static method to return today's date.
of	Overloaded static method to create zoned date time object.
getXXX ()	Used to return various part of ZonedDateTime.
plusXXX()	Add the specified factor and return a ZonedDateTime.
minusXXX()	Subtracts the specified factor and return a ZonedDateTime.
isXXX()	Performs checks on ZonedDateTime and returns Boolean value.
withXXX()	Returns ZonedDateTime with the factor set to the given value.

A ZonedDateTime represents a point in time in a given time zone
that can be converted to an instant on the timeline; it is aware of Daylight Saving Time.

```java
ZonedDateTime currentTime = ZonedDateTime.now();
ZonedDateTime currentTimeInParis = ZonedDateTime.now(ZoneId.of("Europe/Paris"));
System.out.println("India:" + currentTime);
System.out.println("Paris:" + currentTimeInParis);
```



5.6: Date and Time API

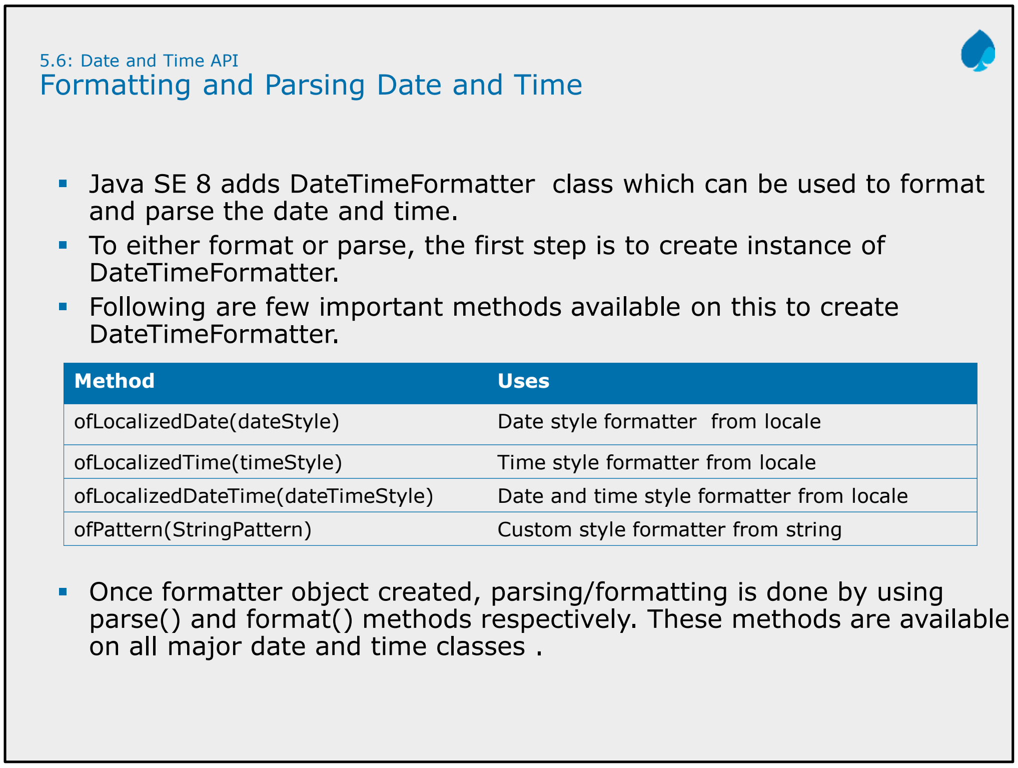
Period and Duration

- The Period class models a date-based amount of time, such as five days, a week or three years.
- Duration class models a quantity or amount of time in terms of seconds and nanoseconds. It is used represent amount of time between two instants.
- Following table shows important and common methods of both:

Method	Uses
between	Use to create either Period or Duration between LocalDates.
of	Creates Period/Duration based on given year, months & days.
ofXXX()	Creates Period/Duration based on specified factors.
getXXX ()	Used to return various part of Period/Duration.
plusXXX()	Add the specified factor and return a LocalDate.
minusXXX(	Subtracts the specified factor and return a LocalDate.
isXXX()	Performs checks on LocalDate and returns Boolean value.
withXXX()	Returns a copy of LocalDate with the factor set to the given value.

The following example shows how to find period between two dates.

```
LocalDate start = LocalDate.of(1947, Month.AUGUST, 15);
LocalDate end = LocalDate.now(); //18/02/2015
Period period = start.until(end);

System.out.println("Days:"+ period.get(ChronoUnit.DAYS));
System.out.println("Months:"+period.get(ChronoUnit.MONTHS));
System.out.println("Years:"+ period.get(ChronoUnit.YEARS));
```



5.6: Date and Time API

Formatting and Parsing Date and Time

- Java SE 8 adds DateTimeFormatter class which can be used to format and parse the date and time.
- To either format or parse, the first step is to create instance of DateTimeFormatter.
- Following are few important methods available on this to create DateTimeFormatter.

Method	Uses
ofLocalizedDate(dateStyle)	Date style formatter from locale
ofLocalizedTime(timeStyle)	Time style formatter from locale
ofLocalizedDateTime(dateTimeStyle)	Date and time style formatter from locale
ofPattern(StringPattern)	Custom style formatter from string

- Once formatter object created, parsing/formatting is done by using parse() and format() methods respectively. These methods are available on all major date and time classes .

To format or parse a date/time, first we need to create instance of DateTimeFormatter. Following example shows, how to format a date using this class. The below example shows how to format the LocalDate in medium style.

```
DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)
LocalDate currentDate = LocalDate.now();
System.out.println(currentDate.format(formatter));
```

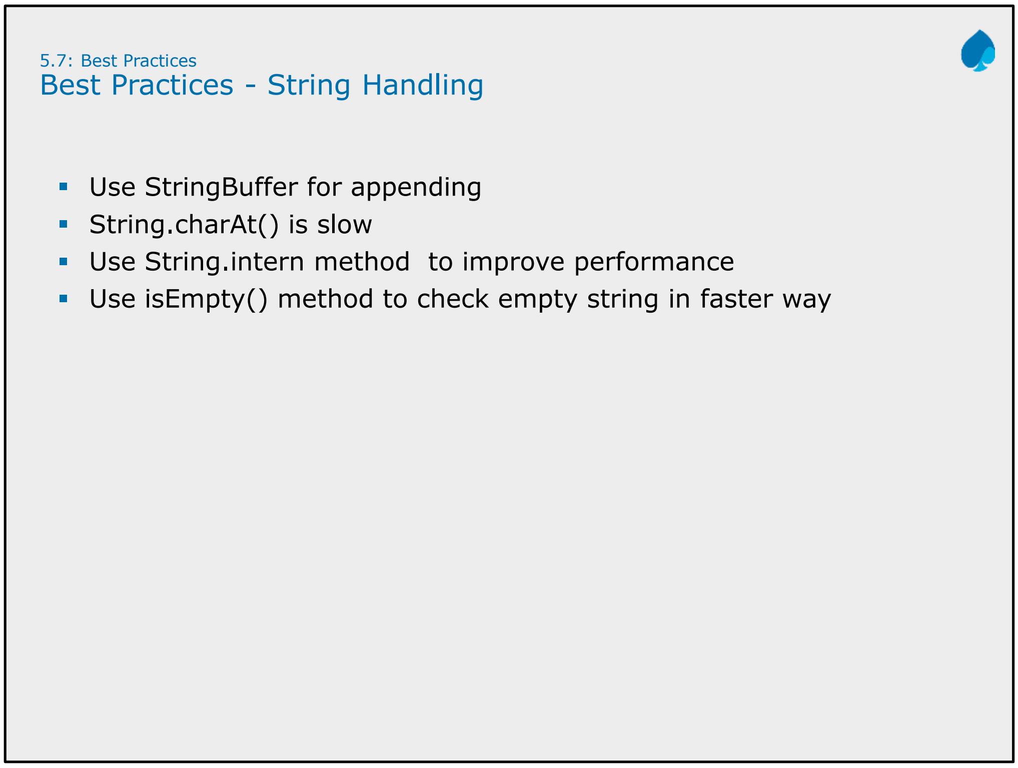
The following example shows how to parse a text string into date.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
String text = "12/02/2015";
LocalDate date = LocalDate.parse(text, formatter);
System.out.println(date);
```

5.6: Date and Time API

Demo

- Execute the following programs:
  - LocalDateDemo.java
  - ZonedDateTimeDemo.java
  - CalculatingPeriod.java
  - FormattingDate.java
  - ParsingDate.java



5.7: Best Practices

Best Practices - String Handling

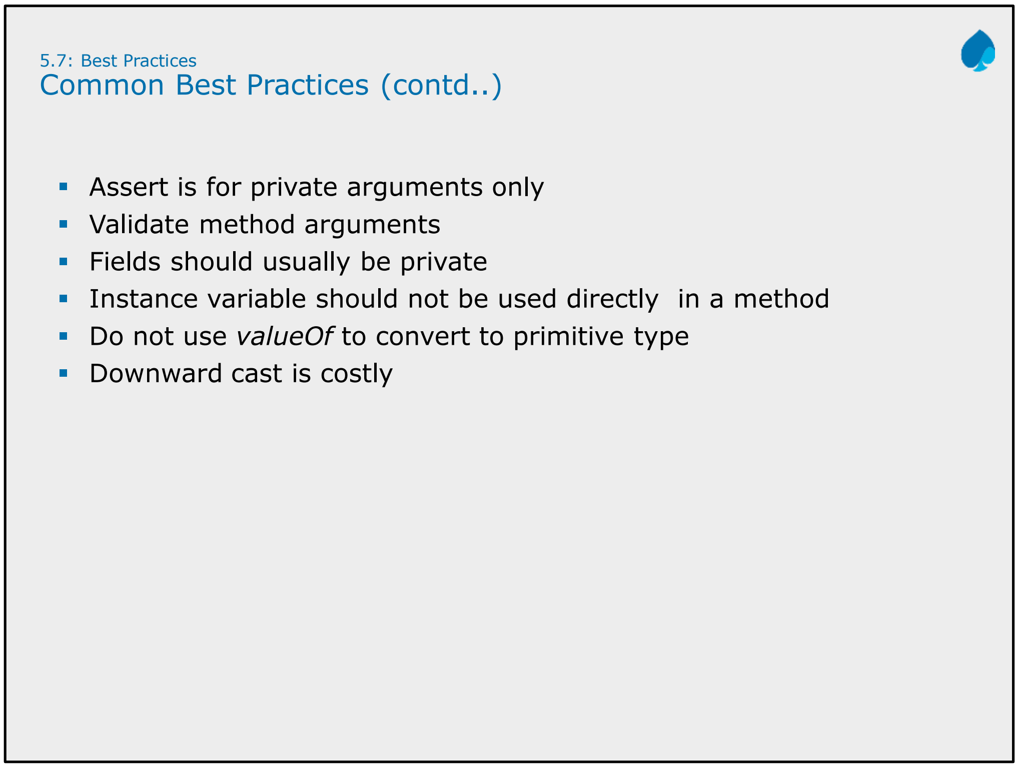
- Use StringBuffer for appending
- String.charAt() is slow
- Use String.intern method to improve performance
- Use isEmpty() method to check empty string in faster way

Use StringBuffer for appending
Always use StringBuffer for appending. Appending by StringBuffer is almost 200 times faster than by using String.concat and "+" operator. But if String constants are to be appended "+" is faster than StringBuffer.append because it is resolved in compile time. Any code with constants is faster.

String.charAt() is slow
The method charAt(int index) returns an individual char from inside a string (see also the substring() method below). The valid index numbers are in the range 0..length-1. Using an index number outside of that range will raise a runtime exception and stop the program at that point.

```
String string = "hello"; char a = string.charAt(0); // a is 'h'
char b = string.charAt(4); // b is 'o‘
char c = string.charAt(string.length() - 1); // same as above line
char d = string.charAt(99); // ERROR, index out of bounds
```

charAt() is slow because it does check for bounds before finding the character. Secondly, it is not declared as final which actually make a bit slower. One can use indexOf and a loop. Which is at least 3 times faster than charAt().



5.7: Best Practices

Common Best Practices (contd..)

- Assert is for private arguments only
- Validate method arguments
- Fields should usually be private
- Instance variable should not be used directly in a method
- Do not use *valueOf* to convert to primitive type
- Downward cast is costly

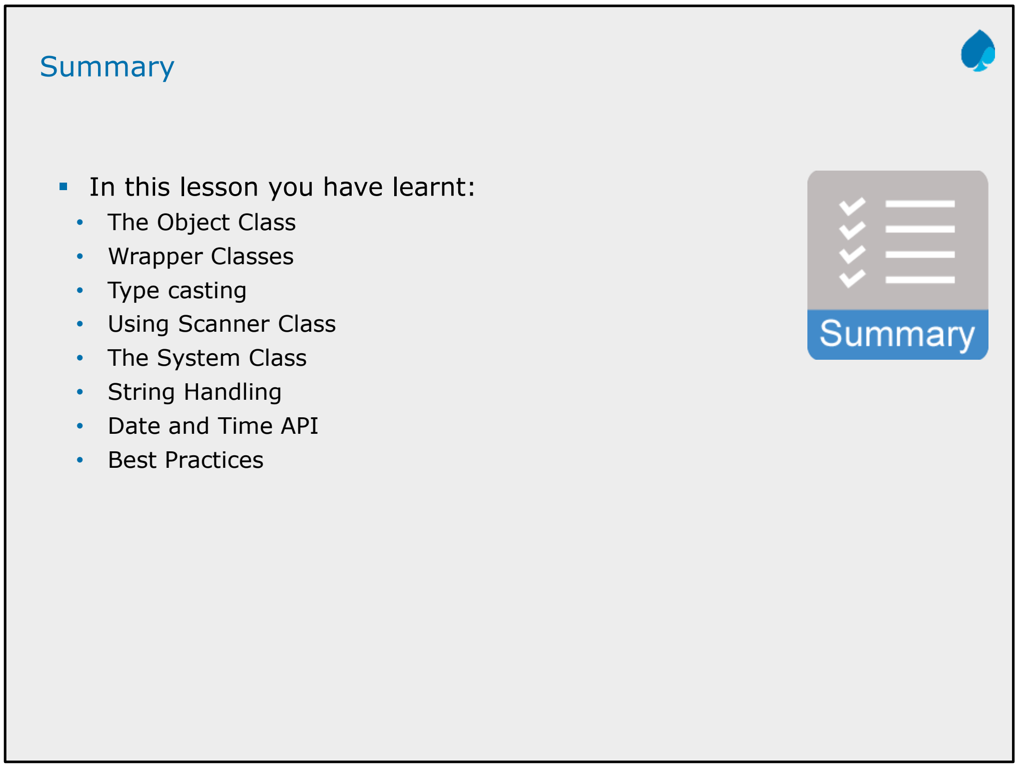
Assert is for private arguments only
Most validity checks in a program are checks on parameters passed to non-private methods. The assert keyword is not meant for these types of validations.
Assertions can be disabled. Since checks on parameters to non-private methods implement the requirements demanded of the caller, turning off such checks at runtime would mean that part of the contract is no longer being enforced.
Conversely, checks on arguments to private methods can indeed use assert. These checks are made to verify assumptions about internal implementation details, and not to check that the caller has followed the requirements of a non-private method's contract.

Validate method arguments
The first lines of a method are usually devoted to checking the validity of method arguments. The idea is to fail as quickly as possible in the event of an error. This is particularly important for constructors. It is a reasonable policy for a class to skip validating arguments of private methods. The reason is that private methods can only be called from the class itself. Thus, a class author should be able to confirm that all calls of a private method are valid. If desired, the assert keyword can be used to check private method arguments, to check the internal consistency of the class.

5.7: Exploring Java Basics

Lab

- Lab 1: Exploring Basic Java Class Libraries

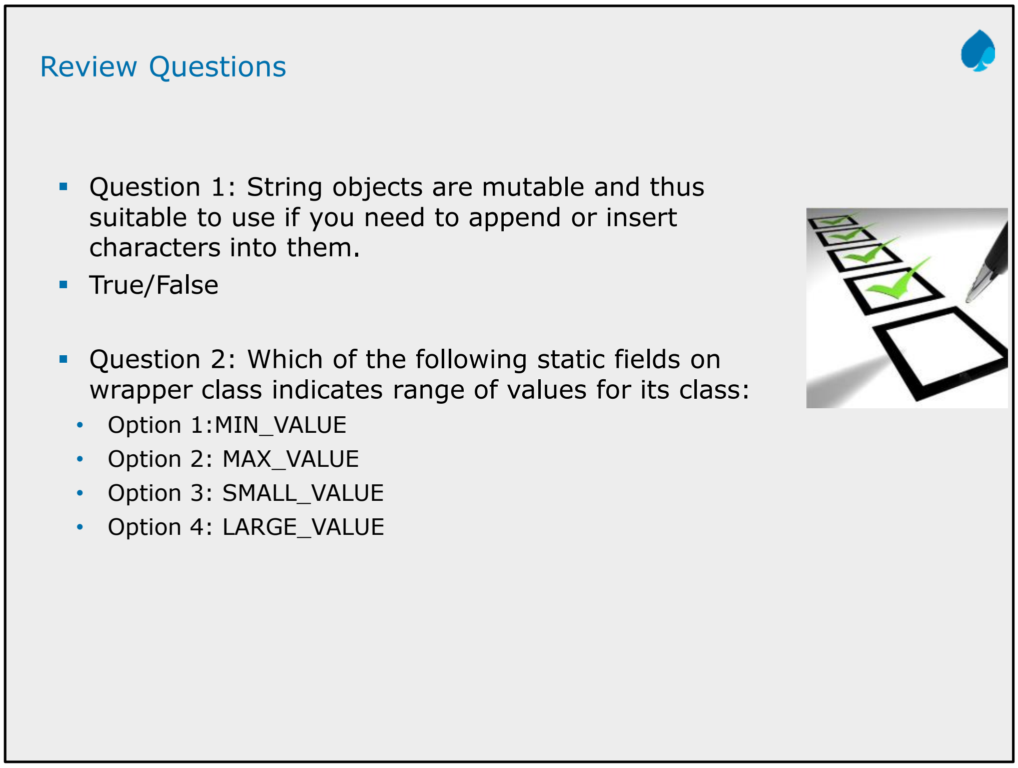


Summary

- In this lesson you have learnt:
  - The Object Class
  - Wrapper Classes
  - Type casting
  - Using Scanner Class
  - The System Class
  - String Handling
  - Date and Time API
  - Best Practices

Summary

Add the notes here.



Review Questions

- Question 1: String objects are mutable and thus suitable to use if you need to append or insert characters into them.
- True/False

- Question 2: Which of the following static fields on wrapper class indicates range of values for its class:
  - Option 1:MIN_VALUE
  - Option 2: MAX_VALUE
  - Option 3: SMALL_VALUE
  - Option 4: LARGE_VALUE