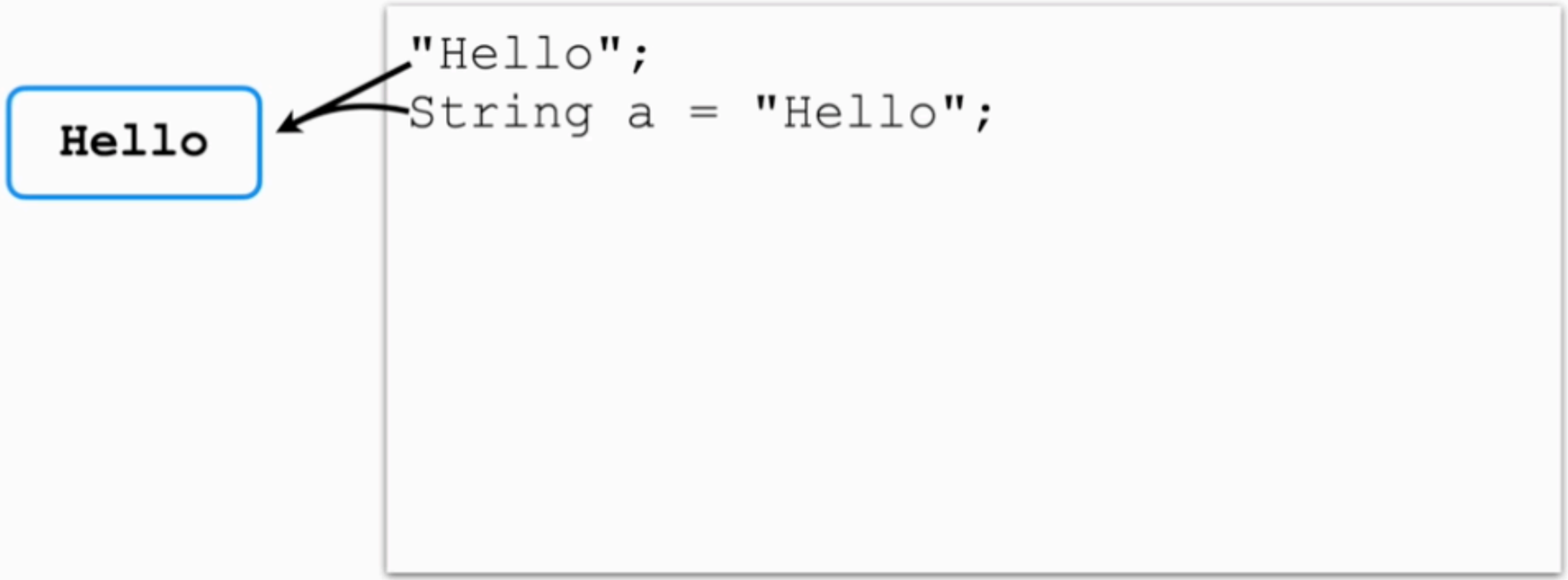
Text, Date, Time, y objetos Numericos

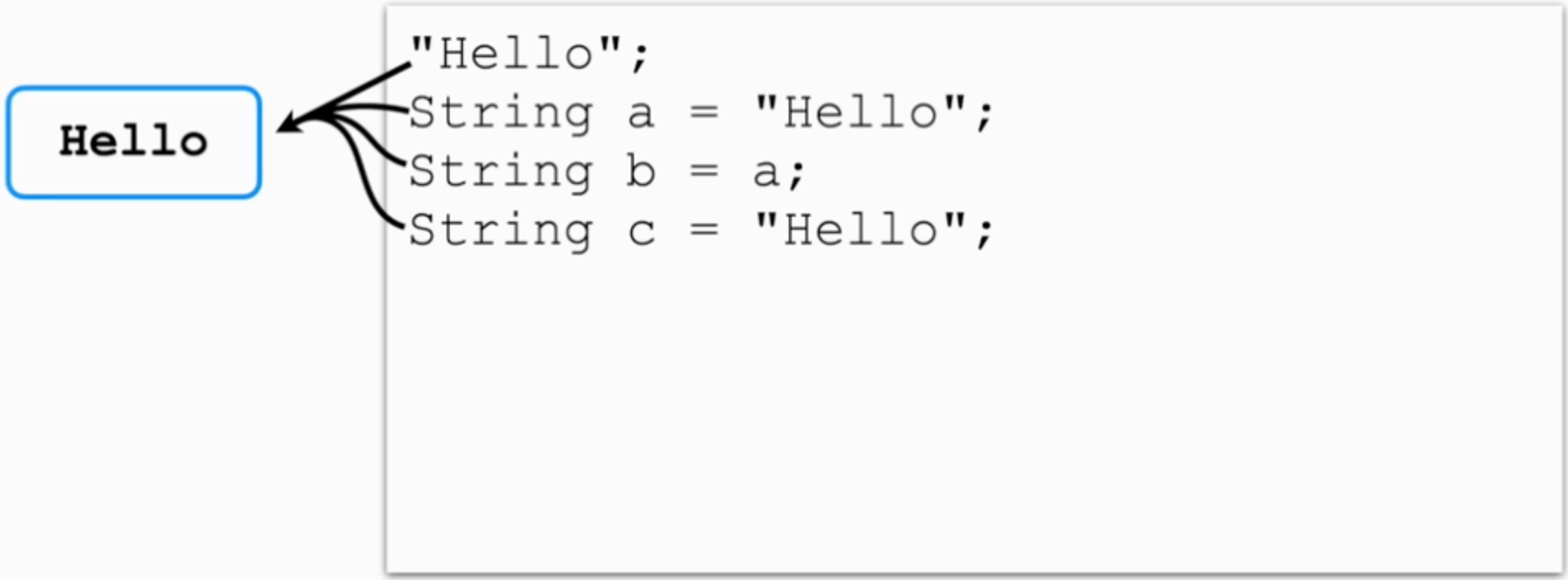
JoeDayz | 2025

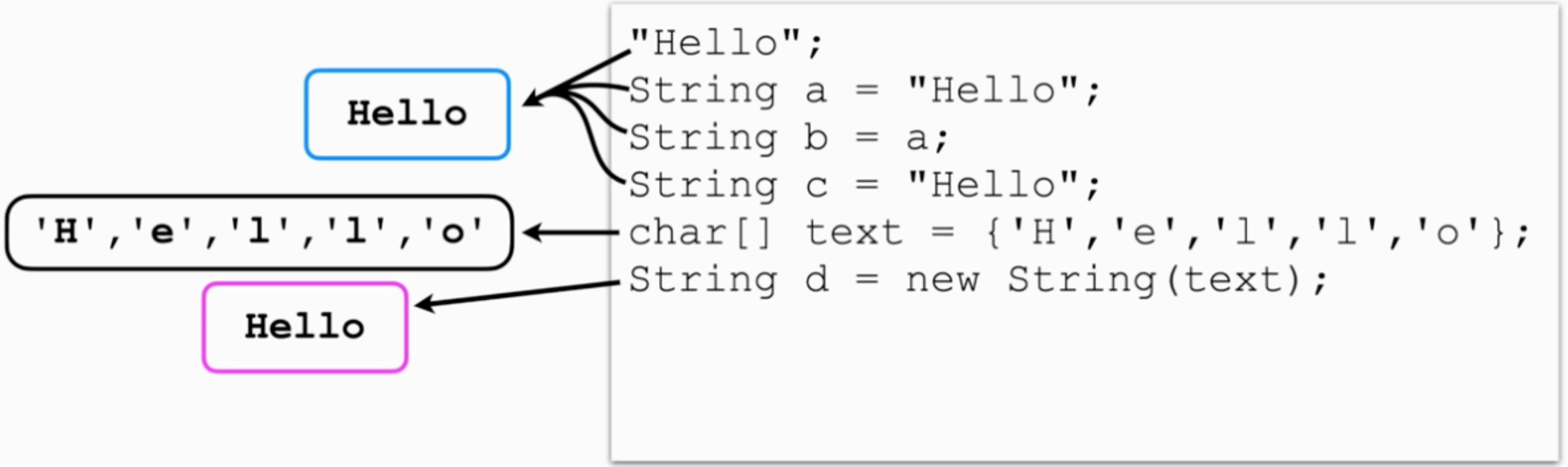
Objectives

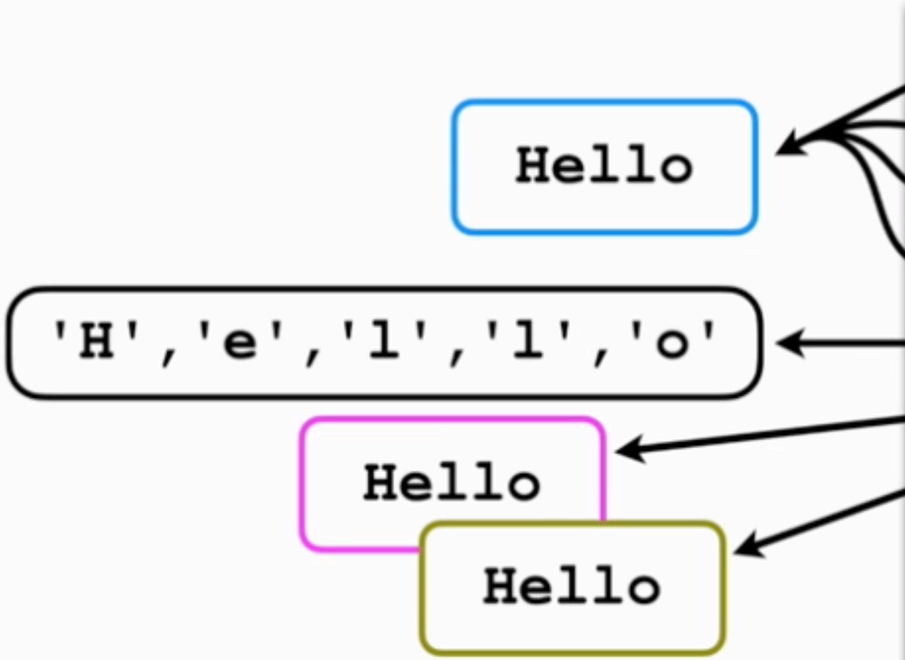
After completing this lesson, you should be able to:

- Manipulate text values by using String, Text Blocks, and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers by using the BigDecimal class
- Handle date and time values
- Describe localization and formatting classes

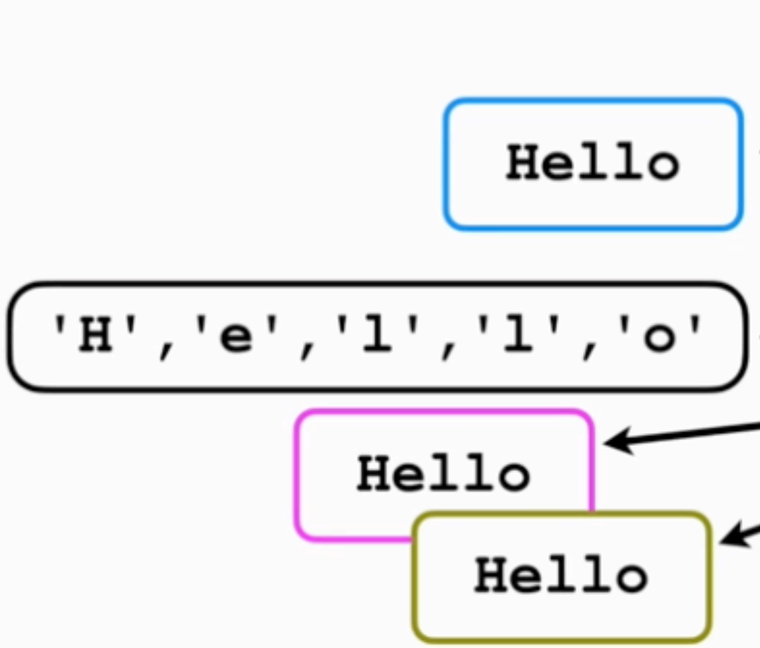

Hello
"Hello";
String a = "Hello";


Hello
"Hello";
String a = "Hello";
String b = a;
String c = "Hello";

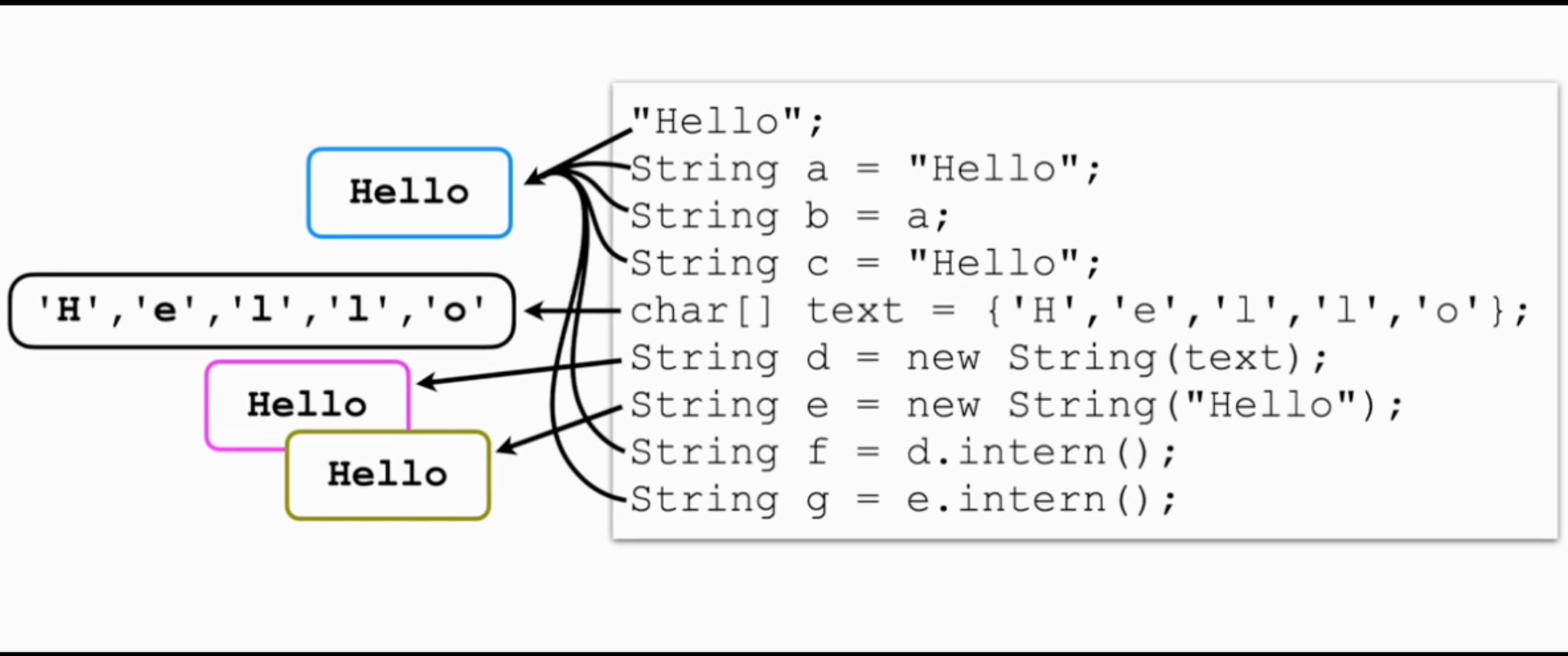

"Hello";
String a = "Hello";
String b = a;
String c = "Hello";
char[] text = {'H','e','l','l','o'};
String d = new String(text);
Hello
'H','e','l','l','o'
Hello




```
"Hello";
String a = "Hello";
String b = a;
String c = "Hello";
char[] text = {'H','e','l','l','o'};
String d = new String(text);
String e = new String("Hello");
```

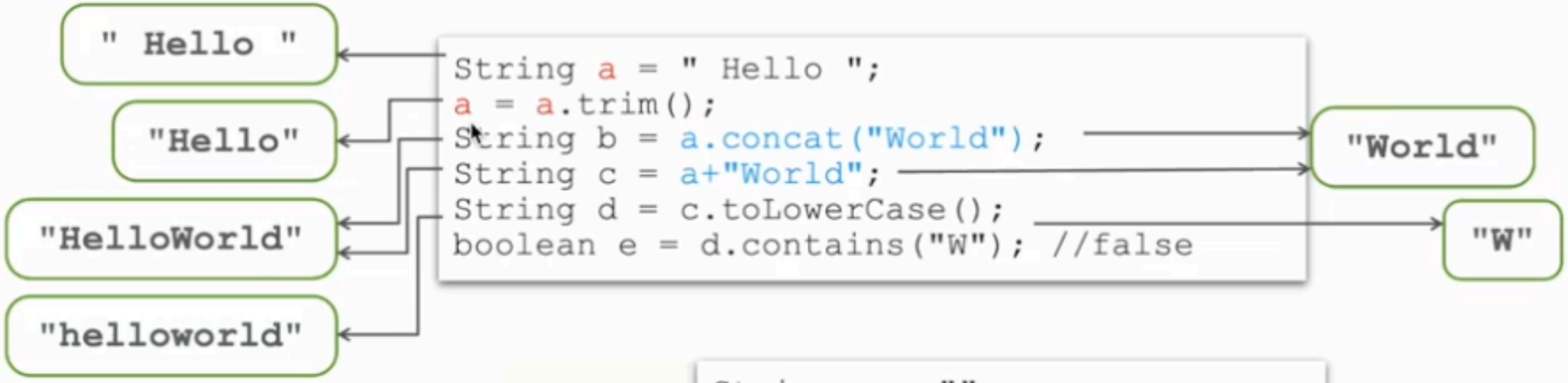



```
"Hello";
String a = "Hello";
String b = a;
String c = "Hello";
char[] text = {'H','e','l','l','o'};
String d = new String(text);
String e = new String("Hello");
String f = d.intern();
```


Hello
'H','e','l','l','o'
Hello
Hello
"Hello";
String a = "Hello";
String b = a;
String c = "Hello";
char[] text = {'H','e','l','l','o'};
String d = new String(text);
String e = new String("Hello");
String f = d.intern();
String g = e.intern();

String Operations

- String objects are immutable. After a string object is initialized, it cannot be changed.
- String operations such as `trim()`, `concat()`, `toLowerCase()`, `toUpperCase()`, and so on would always return a new string, but would not modify the original string.
- It is possible to reassign the string reference to point to another string object.
- For convenience reasons, String allows the use of the + operator instead of the `concat()` method.




```
String s = "";
s=1+1+"u";    // s is 2u
s="u"+1+1;    // s is u11
s="u"+(1+1); // s is u2
```

Note: The + is both a string concatenation as well as an arithmetic operator.

String Indexing

String contains a sequence character indexed by integer.

- The string index starts from `0`.
- When getting a substring of a string, the begin index is inclusive of the result, but the end index is not.
- If a substring is not found, the `indexOf` method returns `-1`.
- Both `indexOf` and `lastIndexOf` operations are overloaded (have more than one version) and accept a char or a string parameter and may also accept a starting index.


```
String a = "HelloWorld";
String b = a.substring(0,5); // b is "Hello"
int c = a.indexOf('o');      // c is 4
int d = a.indexOf('o',5);    // d is 6
int e = a.lastIndexOf('l');  // e is 8
int f = a.indexOf('a');      // f is -1
char g = a.charAt(0);        // g is H
int h = a.length();          // h is 10
char i = a.charAt(10);
```

throws StringIndexOutOfBoundsException

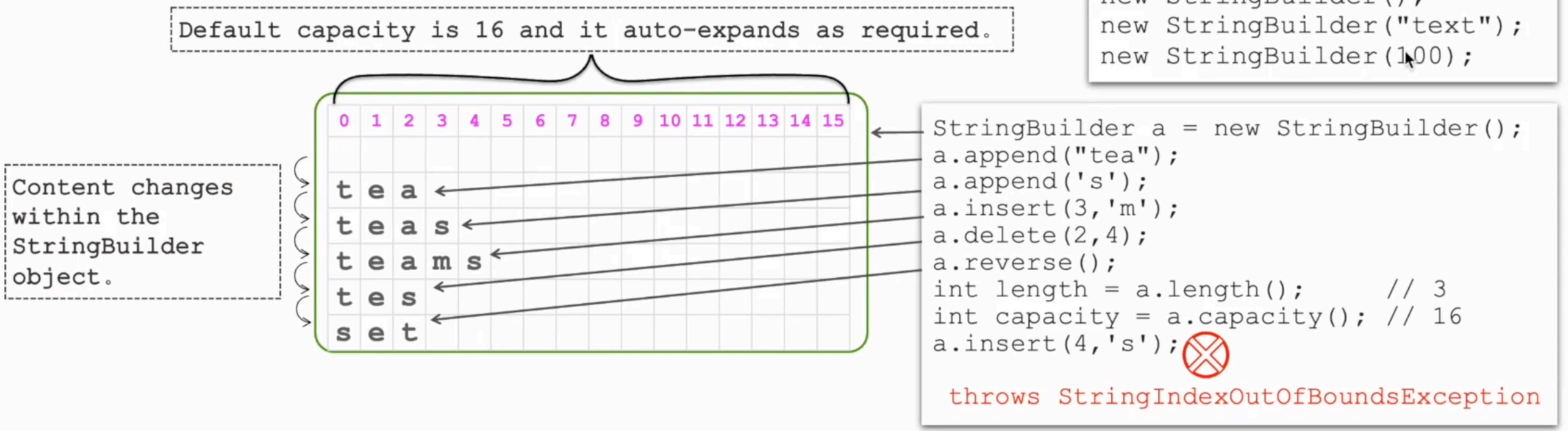
Notes:

- An attempt to access text beyond the last valid index position `length-1` will produce an exception.
- Exception handling is covered later in the course.

Mutable Text Objects

`java.lang.StringBuilder` objects are mutable, allowing modifications of the text they represent.

- Some methods such as substring, `indexOf`, `charAt` are identical to that of a String class.
- Extra methods are available: `append`, `insert`, `delete`, `reverse`.
- The sequence of characters must be continuous. It may contain spaces, but you may not leave gaps of no characters at all.
- Are instantiated using the `new` keyword and be initialized with a predefined content or capacity.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
s	e	t													

```
StringBuilder a = new StringBuilder();
a.append("tea");
a.append('s');
a.insert(3,'m');
a.delete(2,4);
a.reverse();
int length = a.length();      // 3
int capacity = a.capacity(); // 16
a.insert(4,'s');
```

throws StringIndexOutOfBoundsException 

Text Blocks

A text block is a preformatted multiline string literal:

- Begins with """ three double-quote characters followed by a line terminator
- Does not require an explicit escape sequences such as \n or \t
- Does not require you to escape the \" and \' embedded double quotation marks
- Is otherwise treated like any other string object

```
String basicText = "Tea\tprice 1.99 quantity 2\n\"English Breakfast\"\n";

String textBlock = """
                Tea   price 1.99 quantity 2
                "English Breakfast"
                """;

// All String operations apply as usual:
int p1 = basicText.indexOf("price");
int p2 = textBlock.indexOf("price");
```

Spaces, Lines, and Quotes

Text blocks retain left-side spaces, but by default remove line-end spaces.
Extra escape sequences exist for text blocks:

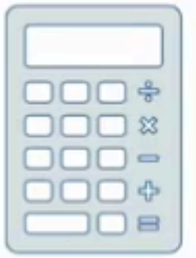
- `\s` inserts a space (for example for padding spaces before the end of the line).
- `\<line terminator>` prevents automatic line break.
- `\"""` represents three quotation marks.

```
String textBlock = """
   This game is a square:
X|0|0
 |0|\s
X|X|0
   Text on the\
   same line
   These are \""" three quotes\
""";
```

```
   This game is a square:
X|0|0
 |0|
X|X|0
   Text on the   same line
   These are """ three quotes
```

✣ **Note:** Avoid overusing escape sequences to improve readability.

Wrapper Classes for Primitives



Wrapper classes apply object-oriented capabilities to primitives.

- A wrapper class is capable of holding a primitive value provided for every Java primitive.
- Construct a wrapper object out of primitive or string by using the `valueOf()` methods.
- Extract primitive values out of the wrapper by using the `xxxValue()` methods.
- Instead of formal conversion of wrapper to primitive and back, you can use direct assignment known as auto-boxing and auto-unboxing.
- Create a wrapper or primitive out of the string by using the `parseXXX()` methods.
- Convert a primitive to a string by using the `String.valueOf()` method.
- Wrapper classes provide constants, such as min and max values for every type.

Notes:

- Advanced text formatting and parsing is covered later.
- Avoid too many auto-boxing and auto-unboxing operations for performance reasons.

```
int a = 42;
Integer b = Integer.valueOf(a);
int c = b.intValue();
b = a;
c = b;
String d = "12.25";
Float e = Float.valueOf(d);
float f = Float.parseFloat(d);
String g = String.valueOf(a);
Short.MIN_VALUE;
Short.MAX_VALUE;
```

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

Representing Numbers Using `BigDecimal` Class

The `java.math.BigDecimal` class is useful in handling decimal numbers that require exact precision.

- All primitive wrappers and `BigDecimal` are immutable and signed.
- However, unlike other numeric wrapper classes, `BigDecimal` has arbitrary precision.
- It is designed to work specifically with decimal numbers and provide convenient `scale` and `round` operations.
- It provides arithmetic operations as methods such as `add`, `subtract`, `divide`, `multiply`, `remainder`.
- It is typically used to represent decimal numbers that require exact precision, such as fiscal values.

```
BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);                  // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

Method Chaining

- When an operation returns an object, you may invoke the next operation on this object immediately.
- It is possible to use the chain method invocations technique with any operation that returns an object.

Examples:

- Text manipulating operations of `String` return `String` objects.
- Arithmetic operations of `BigDecimal` return `BigDecimal` objects.

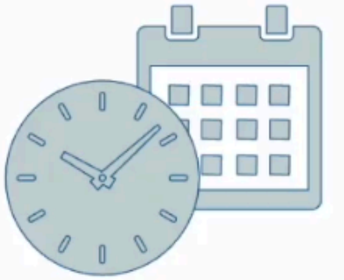
```
String s1 = "Hello";
Strung s2 = s1.concat("World").substring(3,6); // s2 is "loW"

BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);                    // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

Notes:

- Method chaining is a common coding technique used when an intermediate object returned by a method is not required, so code immediately invokes of next method and so on.
- Without method chaining, code appears to be cluttered with unnecessary intermediate variables:

```
BigDecimal taxedPrice = price.add(tax);
price = taxedPrice.setScale(2,RoundingMode.HALF_UP);
```

Local Date and Time API



Handle date and time values with the API provided with the `java.time` package.

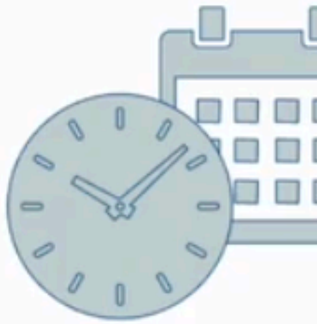
- Implemented with `LocalDate`, `LocalTime`, and `LocalDateTime` classes.
- Date and time objects can be created using the following methods:
  - `now()` to get current date and time, or
  - `LocalDateTime.of(year, month, day, hours, minutes, seconds, nanoseconds)`
  - `LocalTime.of(hours, minutes, seconds, nanoseconds)`
  - `LocalDate.of(year, month, day)` for specific date and time or
  - Combine other date and time objects by using `atTime()` and `of(localDate, localTime)` or
  - Extract date and time portions from `LocalDateTime` by using `toLocalDate()` and `toLocalTime()`.

```
LocalDate today = LocalDate.now();
LocalTime thisTime = LocalTime.now();
LocalDateTime currentDateTime = LocalDateTime.now();
LocalDate someDay = LocalDate.of(2019,Month.APRIL,1);
LocalTime someTime = LocalTime.of(10,6);
LocalDateTime otherDateTime = LocalDateTime.of(2019,Month.MARCH,31,23,59);
LocalDateTime someDateTime = someDay.atTime(someTime);
LocalDate whatWasTheDate = someDateTime.toLocalDate();
```

Notes:

- Date and Time API (`java.time` package) was introduced in Java SE 8.
- Earlier Java versions used the `java.util.Date` class to represent date and time values.

More Local Date and Time Operations



Characteristics of local date and time operations:

- Local date and time objects are immutable.
- All date and time manipulation methods will produce new date and time objects.
- New date and time objects can be produced out of existing objects using `plusXXX()` or `minusXXX()` or `withXXX()` methods.
- It is possible to chain method invocations together, because all date and time manipulation operations return da and time objects.
- Individual portions of date and time objects can be retrieved with `getXXX()` methods.
- Operations `isBefore()` and `isAfter()` check if a date or time is before or after another.

```
LocalDateTime current = LocalDateTime.now();
LocalDateTime different =  current.withMinute(14).withDayOfMonth(3).plusHours(12);
int year = current.getYear();
boolean before = current.isBefore(different);
```

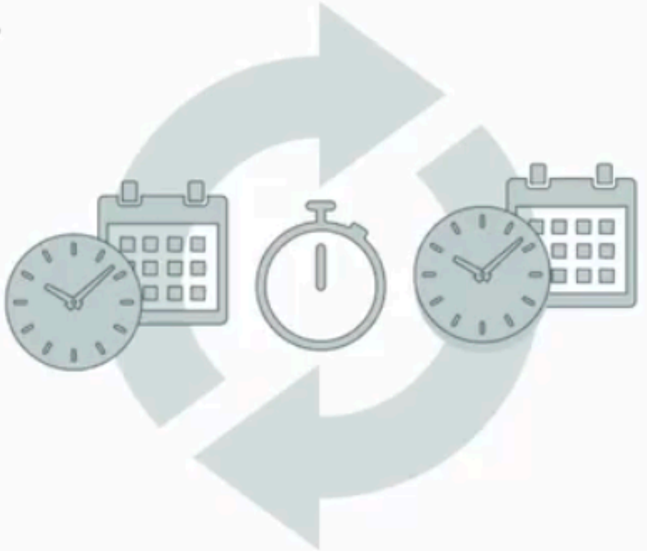
✣ **Reminder:** Method chaining helps to avoid cluttering code with unnecessary intermediate variables.

Instants, Durations, and Periods

Handle periods and durations of time with the following classes from the `java.time` package:

- `Duration`: Represents an amount of time in nanoseconds
- `Period`: Represents an amount of time in units such as years or days
- `Instant`: Represents an instantaneous point on the timeline (time stamp)
- Produce with `now()`, `ofXXX()`, `plusXXX()`, `minusXXX()`, `withXXX()` methods.
- Specific time values can be retrieved with `getXXX()` methods.
- Measure distance between points in time with `between()`, `isNegative()` methods.
- Use identical coding techniques such as method chaining.

```
LocalDate today = LocalDate.now();
LocalDate foolsDay = LocalDate.of(2019, Month.APRIL, 1);
Instant timeStamp = Instant.now();
int nanoSecondsFromLastSecond = timeStamp.getNano();
Period howLong = Period.between(foolsDay, today);
Duration twoHours = Duration.ofHours(2);
long seconds = twoHours.minusMinutes(15).getSeconds();
int days = howLong.getDays();
```



Notes:

- Instant and duration are more suitable for implementing system tasks, such as using a time stamp when writing logs.
- Period is more suitable for implementing business logic.
- Just like the rest of the local date and time API, duration, period, and instant objects are immutable.

Zoned Date and Time

Time zones can be applied to local date and time values.

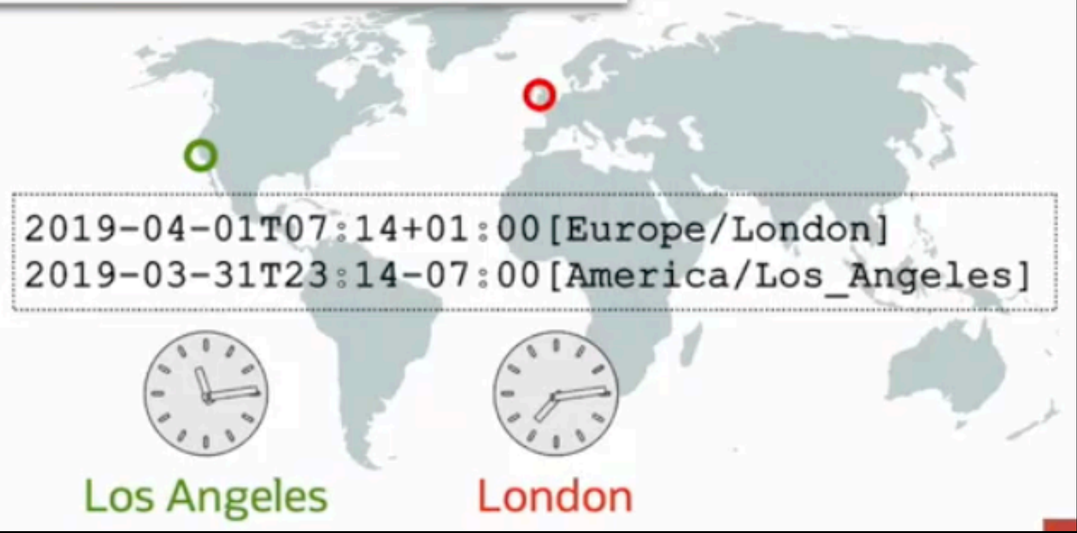
`java.time.ZonedDateTime` class:

- Represents date and time values according to time zone rules, such as daylight saving time and time differences
- Has the same time management capabilities as `LocalDateTime` objects
- Provides time zone specific operations such as a `withZoneSameInstant` method

```
ZoneId london = ZoneId.of("Europe/London");
ZoneId la  = ZoneId.of("America/Los_Angeles");
LocalDateTime someTime = LocalDateTime.of(2019,Month.APRIL,1,07,14);
ZonedDateTime londonTime = ZonedDateTime.of(someTime, london);
ZonedDateTime laTime = londonTime.withZoneSameInstant(la);
```

The `java.time.ZoneId` class defines time zones:

- `ZoneId.of("America/Los_Angeles");`
- `ZoneId.of("GMT+2");`
- `ZoneId.of("UTC-05:00");`
- `ZoneId.systemDefault();`



Representing Languages and Countries

Make your application adjustable to different languages and locations around the world.

- `java.util.Locale` class represents languages and countries.
- ISO 639 language and ISO 3166 or country codes or UN M.49 area codes are used to set up locale objects.
- Locale can represent just language or a combination of language plus country or area.
- Variant is an optional parameter, designed to produce custom locale variations.
- Language tag string allows constructing locales for various calendars, numbering systems, currencies, and so on.

```
                                                    Language Country Variant
Locale uk = Locale.of("en","GB");          // English  Britain
Locale th = Locale.of("th","TH","TH");     // Thai     Thailand  (Thai digits variant)
Locale us = Locale.of("en","US");          // English  America
Locale fr = Locale.of("fr","FR");          // French   France
Locale cf = Locale.of("fr","CA");          // French   Canada
Locale fr = Locale.of("fr","029");         // French   Caribbean
Locale es = Locale.of("fr");               // French
Locale current = Locale.getDefault();       // current default locale
// Example constructing locale that uses Thai digits and Buddhist calendar:
Locale th = Locale.forLanguageTag("th-TH-u-ca-buddhist-nu-thai");
```

Formatting and Parsing Numeric Values

The `java.text.NumberFormat` class is used to parse and format numeric values.

- Works with any Java number, including primitives, wrapper classes, and BigDecimal objects.

```
BigDecimal price = BigDecimal.valueOf(2.99);
Double tax = 0.2;
int quantity = 12345;
Locale locale = Locale.of("en","GB");
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
String formattedPrice = currencyFormat.format(price);
String formattedTax = percentageFormat.format(tax);
String formattedQuantity = numberFormat.format(quantity);
```

value initializations

locale initialization

formatter initializations

formatting values

```
£2.99 20% 12,345
```

formatted results

- Method parse return type is Number and it can be converted to numeric primitive wrappers or BigDecimal types.

```
BigDecimal p = BigDecimal.valueOf((Double)currencyFormat.parse("£1.7"));
Double t = (Double)percentageFormat.parse("12%");
int q = numberFormat.parse("54,321").intValue();
```

parsing values

```
1.75 0.12 54321
```

parsed values

Formatting and Parsing Date and Time Values

- `java.time.format.DateTimeFormatter` parses and formats date and time values.
- `java.time.format.FormatStyle` enum defines standard format patterns.
- Alternatively, set up custom format patterns.

```
LocalDate date = LocalDate.of(2019,Month.APRIL,1);
Locale locale = Locale.of("en","GB");
DateTimeFormatter dateTimeFormat =
  DateTimeFormatter.ofPattern("EEEE dd MMM yyyy",locale);
String result = date.format(dateTimeFormat);
             // dateTimeFormat.format(date);
```

```
Monday 01 Apr 2019
```

value initialization

locale initialization

formatter initialization

format value

formatted result

```
date = LocalDate.parse("Tuesday 31 Mar 2020",dateTimeFormat);
locale = Locale.of("ru");
dateTimeFormat =
  DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM)
                   .localizedBy(locale);
result = date.format(dateTimeFormat);
      // dateTimeFormat.format(date);
```

```
31 мар. 2020 г.
```

parse value

reset locale

reset formatter

format value

formatted result

Localizable Resources

The `java.util.ResourceBundle` class represents bundles of localizable resources.

- Resources may be stored as classes or plain text files with the `.properties` extension.
- Resources are placed into resource bundles as `<key>=<value>` pairs.
- They may represent user-visible messages or message patterns with substitution parameters.
- A default bundle can be used if no locale is specified, or if the resource you're trying to get is not present in a language and country-specific bundle.

```
Locale locale = Locale.of("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resources.messages",locale);
String helloPattern = bundle.getString("hello");
String otherMessage = bundle.getString("other");
```

```
resources          (package folder)
  messages.properties
  messages_en_GB.properties
  messages_ru.properties
```

```
hello=こんにちは {0}
product={0}, 価格 {1}, 分量 {2}, 賞味期限は {3}
other=他に何か
```

Default bundle can be in any language.

```
hello=Hello {0}
product={0}, price {1}, quantity {2}, best before {3}
```

```
hello=Привет {0}
product={0}, цена {1}, количество {2}, годен до {3}
```

✣ **Note:** Value substitutions are explained later.

Formatting Message Patterns

`java.text.MessageFormat` class is used to:

- Format messages by substituting values into text patterns
- Parse messages by extracting values out of the text based on a pattern

Message patterns are typically stored in resource bundles.

```
product={0}, price {1}, quantity {2}, best before {3}
```

`resources/messages_en_GB.properties`

```
Locale locale = Locale.of("en","GB");
ResourceBundle bundle =
  ResourceBundle.getBundle("resources.messages",locale);
// assume following values are already formatted as String objects:
// name, price, quantity, bestBefore
String pattern = bundle.getString("product");
String result = MessageFormat.format(pattern,
                                     name,price,quantity,bestBefore);
```

initialize locale and bundle

prepare values

get pattern

substitute values

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```

formatted result

Note: See notes for more examples and use cases.

Formatting and Localization: Summary

resources/messages_en_GB.properties

```
product={0}, price {1}, quantity {2}, best before {3}
```

```
Locale locale = Locale.of("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resources.messages",locale);
String pattern = bundle.getString("product");

String name = "Cookie";
BigDecimal price = BigDecimal.valueOf(2.99);
int quantity = 4;
LocalDate bestBefore = LocalDate.of(2019,Month.APRIL,1);

NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("dd MMM yyyy",locale);

String fPrice = currencyFormat.format(price);
String fQuantity = numberFormat.format(quantity);
String fBestBefore = dateFormat.format(bestBefore); // or bestBefore.format(dateFormat);

String result = MessageFormat.format(pattern, name, fPrice, fQuantity, fBestBefore);
```

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```

- **Note:** See notes for an alternative example.

Summary

In this lesson, you should have learned how to:

- Manipulate text values by using String, Text Blocks, and StringBuilder classes
- Describe primitive wrapper classes
- Perform string and primitive conversions
- Handle decimal numbers by using the BigDecimal class
- Handle date and time values
- Describe localization and formatting classes

Quiz

1. Which three statements are true?

- [ ] You can instantiate a StringBuilder with a predefined content or capacity.
- [ ] StringBuilder objects automatically expand capacity as needed.
- [ ] StringBuilder objects are mutable.
- [ ] StringBuilder objects are thread-safe.

2. Which is NOT true about Strings in Java?

- ○ Always create a single copy of each string literal
- ○ Are a class
- ○ Allow to modify text value stored in a String object
- ○ Represent a sequence of characters

3. Which is NOT true about Format classes?

- ○ They are used to format text.
- ○ Parse operations return null when text is incorrectly formatted.
- ○ They are used to parse text.
- ○ They work with text, numeric, date, and time values.