7

Manipulating and Formatting the Data in Your Program

Objectives

After completing this lesson, you should be able to:

- Describe the `String` class and use some of the methods of the `String` class
- Use the JDK API documentation to search for and learn how to use a class
- Describe the `StringBuilder` class
- Explain what a constant is and how to use it
- Explain the difference between promoting and casting of variables

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

`String` Class

```
String hisName = "Fred Smith";
String herName = new String("Anne Smith");
```

Standard syntax

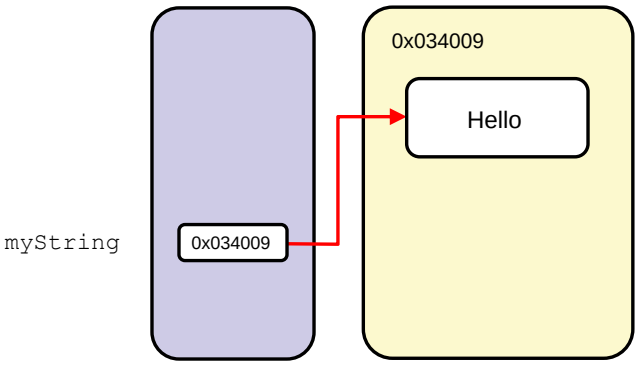
The `new` keyword can be used, but it is *not* best practice:

- A `String` object is immutable; its value cannot be changed.
- A String object can be used with the string concatenation operator symbol (+) for concatenation.

- The `String` class is one of the many classes included in the Java class libraries. The `String` class provides you with the ability to store a sequence of characters. You will use the `String` class frequently throughout your programs. Therefore, it is important to understand some of the special characteristics of strings in the Java programming language. Because a `String` object is immutable, its value cannot be changed. (There are technical reasons, beyond the scope of this course, as to why this immutability is useful. One simple example is that this immutability ensures that a `String` can be used by several different classes safely because it cannot be changed.)
- Creating a `String` object using the `new` keyword creates two `String` objects in memory, whereas creating a `String` object by using a string literal creates only one object; therefore, the latter practice is more memory-efficient. To avoid the unnecessary duplication of `String` objects in memory, create `String` objects without using the `new` keyword.

Concatenating Strings

```
String myString = "Hello";
```

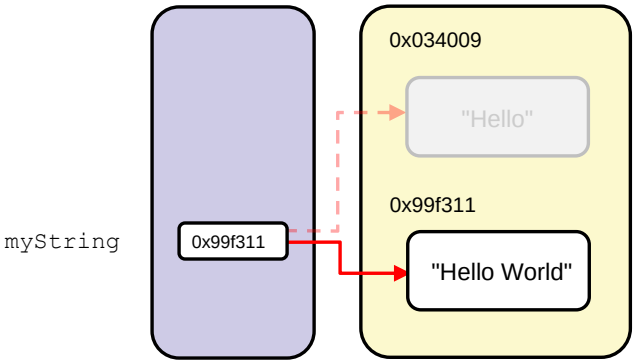


Because `String` is immutable, concatenating two strings requires creating a new string.

The diagram shows a `String` object containing the string "`Hello`".

Concatenating Strings

```
String myString = "Hello";
myString = myString.concat(" World");
```

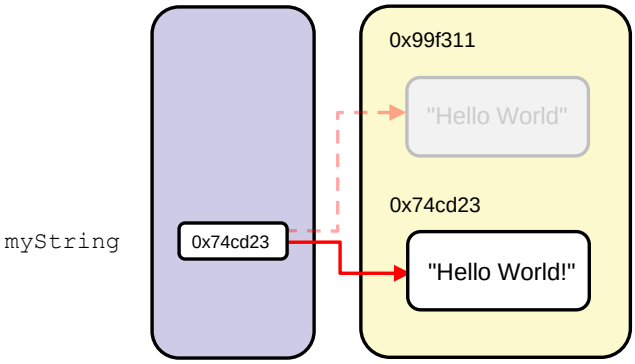


Here is the string " `World`" being concatenated to the original string. The `concat` method is being used here, but whether you use that or the concatenation operator (`+`), a new `String` object is created and a new `String` reference is returned that points to this new object.

In the diagram, this is shown by the fact that the `String` reference `myString` is no longer 0x034009, and because that object is no longer referred to, it is now inaccessible and will be garbage collected.

Concatenating Strings

```
String myString = "Hello";
myString = myString.concat(" World");
myString = myString + "!"
```



Finally, on concatenating another string, this time using the concatenation operator, the same thing happens again. A new object is created and the reference for this object is assigned to `myString`.

`String` Method Calls with Primitive Return Values

A method call can return a single value of any type.

- An example of a method of primitive type `int`:

```
String hello = "Hello World";
int stringLength = hello.length();
```

Like most classes, the `String` class has a number of useful methods. Almost all of these methods do their useful work by returning a single value (Java allows only a single return from a method). The return type (essentially the type of the method) can be a primitive or a reference to an object.

To be able to use the return value in your code, you will typically use the assignment operator to assign the value (or reference) to a type that you have declared for this purpose.

The example in the slide shows the use of reference `hello` to call the method `length`. Because the object this reference refers to is the string `Hello World`, this method call will return the value `11` and place it in the variable `stringLength`. `int` is the type of the method `length`.

`String` Method Calls with Object Return Values

Method calls returning objects:

```
String greet = " HOW ".trim();
String lc = greet + "DY".toLowerCase();
```

Or

```
String lc = (greet + "DY").toLowerCase();
```

This example shows several method calls that return object references.

First, the `String` object `"  HOW  "` is instantiated and has the method `trim` called on it. Because a string literal returns an object reference, this is exactly the same as calling the method `trim` on the reference. Notice that the string `"  HOW  "` has two spaces on either side of the word. The string returned will be just three characters long because these spaces will be removed. This new string will be referenced by `greet`.

The next example shows a method call not being assigned to a type, but simply used in an expression. The method `toLowerCase` is called on the string `"DY"`, returning `"dy"`. `lc` now references an object containing `"HOWdy"`.

Finally, note how an alternative version with parentheses ensures that the two strings are concatenated (creating a new string) before `toLowerCase` is called. `lc` now references an object containing `"howdy"`.

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Java API Documentation

Consists of a set of webpages;

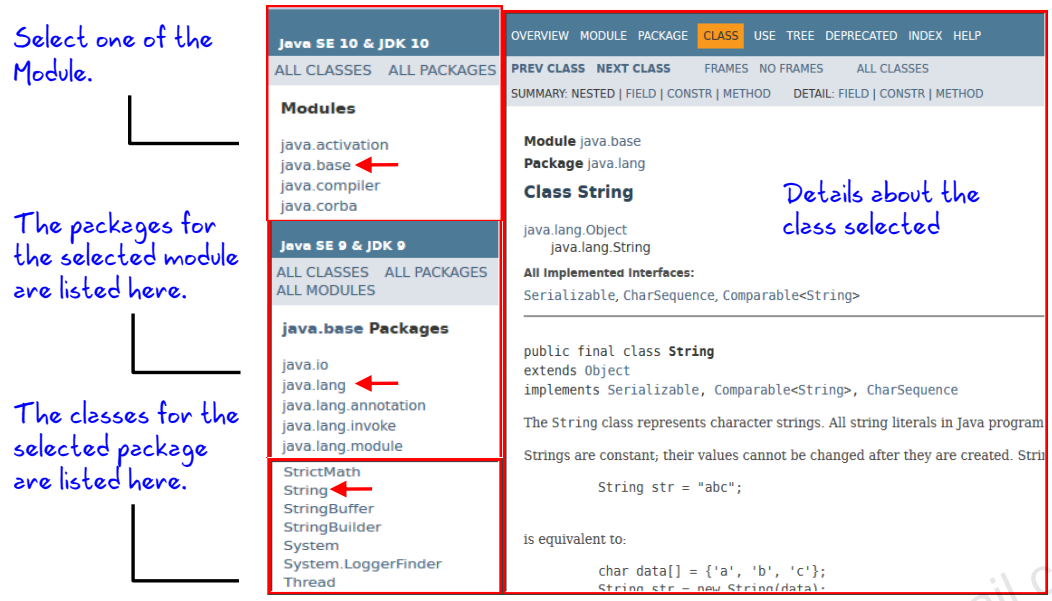
- Lists all the classes in the API
  - Descriptions of what the class does
  - List of constructors, methods, and fields for the class
- Highly hyperlinked to show the interconnections between classes and to facilitate lookup
- Available on the Oracle website at: https://docs.oracle.com/javase/10/docs/api/overview-summary.html

All of the Java technology JDKs contain a series of prewritten classes for you to use in your programs. These Java technology class libraries are documented in the Java API documentation for the version of the JDK that you are using. The class library specification is a series of HTML webpages that you can load in your web browser.

A Java class library specification is a very detailed document outlining the classes in the API. Every API includes documentation describing the use of classes and their fields and methods. When you are looking for a way to perform a certain set of tasks, this documentation is the best source for information about the classes in the Java class libraries.

You learn more about constructors in the “Using Encapsulation” lesson.

JDK 10 API Documentation



In the screenshot in the slide, you can see the three main panels of the webpage.

Modules are a new programming construct introduced in JDK 9 and you will learn more about modules in Lesson 16 of this course.

The top-right panel allows you to select a module. The packages of a particular module are listed. You can select a class from a particular package. But if you do not know the package of a particular class, you can select All Classes.

The bottom-left panel gives the list of packages in a module, you can then select a particular package and then all the classes in that are listed.

In this panel, the class `String` has been selected, populating the main panel on the right with the details of the class `String`. The main panel on the right contains a lot of information about the class, so you need to scroll down to access the information you need.

Java Platform SE and JDK Version 10 API Specification

This document is divided into three sections:

- **Java SE:**
  - The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing.
  - These APIs are in modules whose names start with java.
- **JDK**
  - The Java Development Kit (JDK) APIs are specific to the JDK and will not necessarily be available in all implementations of the Java SE Platform.
  - These APIs are in modules whose names start with jdk.
- **JavaFX:**
  - The JavaFX APIs define a set of user-interface controls, graphics, media, and web packages for developing rich client applications.
  - These APIs are in modules whose names start with javafx.

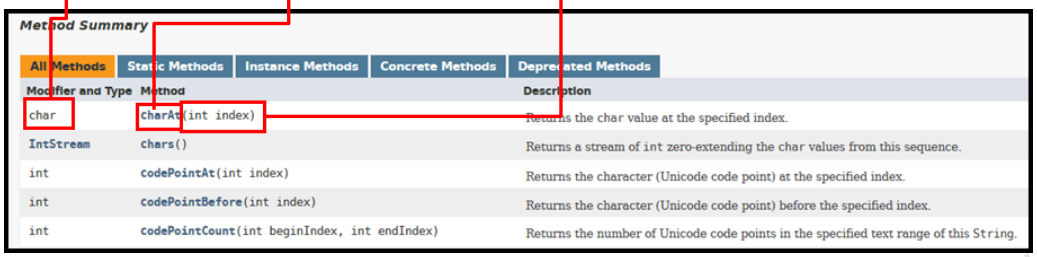
Java Platform SE 10: Method Summary

```
public int charAt(String str)
```

The return type of the method

The name of the method

The type of the parameter that must be passed into the method



Method Summary

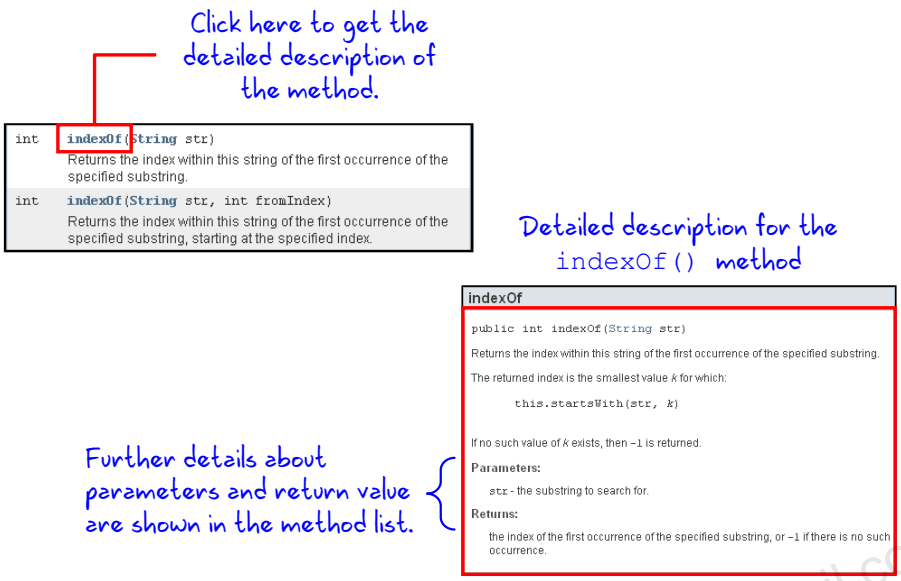
All Methods | Static Methods | Instance Methods | Concrete Methods | Deprecated Methods

Modifier and Type	Method	Description
char	charAt(int index)	Returns the char value at the specified index.
IntStream	chars()	Returns a stream of int zero-extending the char values from this sequence.
int	codePointAt(int index)	Returns the character (Unicode code point) at the specified index.
int	codePointBefore(int index)	Returns the character (Unicode code point) before the specified index.
int	codePointCount(int beginIndex, int endIndex)	Returns the number of Unicode code points in the specified text range of this String.

If you keep scrolling through the details for the `String` class, you will come to the list of methods (only a small subset of this list is shown here).

This master list of methods gives the basic details for the method. In this case, you can see that the name of the method is `charAt`, its type is `char`, and it requires an index (of type `int`) to be passed in. There is also a brief description that this method returns the `char` value at a particular index in the string. For any of the methods, the method name and the parameter types are hyperlinked so that you can get more details.

Java Platform SE 10: Method Detail



For any of the methods, the method name and the parameter types are hyperlinked so that you can get more details. The example here shows the detailed description for one of the `indexOf()` methods of `String`.

`indexOf` Method Example

```
1  String phoneNum = "404-543-2345";
2  int idx1 = phoneNum.indexOf('-');
3  System.out.println("index of first dash: "+ idx1);
4
5
6  int idx2 = phoneNum.indexOf('-', idx1+1);
7  System.out.println("second dash idx: "+idx2);
```

The 1-arg version

The 2-arg version

This example shows how to get the location of the first '-' character by using the 1-arg version of `indexOf`, and then by using the 2-arg version to get the location of the second '-'.

If you wanted to convert the phone number to an `int`, you could do something like this:

1. Find the dashes by using the `indexOf` method (as shown above).
2. Build a new `String` without dashes by using the `substring` method and concatenation.
3. Convert this `String` to an `int` by using the `parseInt` method of `Integer`.

The `parseInt` method of the `Integer` class is covered in the lesson "Using Encapsulation."

Exercise 7-1: Use `indexOf` and `substring` Methods

In this exercise, you get and display a customer's first name.

1. Open the project **`Exercise_07-1`** in NetBeans.
2. Use the `indexOf` method to get the index for the space character (" ") within `custName`. Assign it to `spaceIdx`.
3. Use the `substring` method and the `spaceIdx` to get the first name portion of `custName`.
   - Assign it to `firstName`.
   - Print `firstName`.

In this exercise, you use `indexOf` and `substring` methods to get just the customer's first name and display it.

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

`StringBuilder` Class

`StringBuilder` provides a mutable alternative to `String`. `StringBuilder`:

- Is instantiated using the `new` keyword
- Has many methods for manipulating its value
- Provides better performance because it is mutable
- Can be created with an initial capacity

`String` is still needed because:

- It may be safer to use an immutable object
- A method in the API may require a string
- It has many more methods not available on `StringBuilder`

The `StringBuilder` class is "mutable." This means that it can be changed in place. You will recall that when you modify the value of a `String` variable, a new `String` object is created for the new value. `String` objects are "immutable." A `String` object's value cannot be changed.

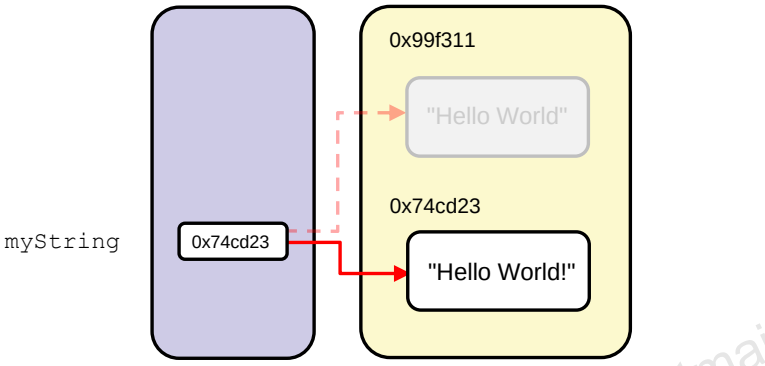
- Unlike `String`, there is no shortcut to instantiate a `StringBuilder`. It is simply instantiated like any other object by using the `new` keyword.
- A small sampling of the `StringBuilder` methods for manipulation of data values are: `append`, `delete`, `insert`, and `replace`.
- `StringBuilder` provides better performance because it does not create new objects in memory whenever a change is made. Performance is also benefited whenever you can set an initial capacity for the object, as opposed to letting it grow and allocate memory dynamically.
- `StringBuilder` is not a complete replacement for `String`, but it is more suitable if many modifications are likely to be made to its value.

`StringBuilder` Advantages over String for Concatenation (or Appending)

String Concatenation

- Costly in terms of creating new objects

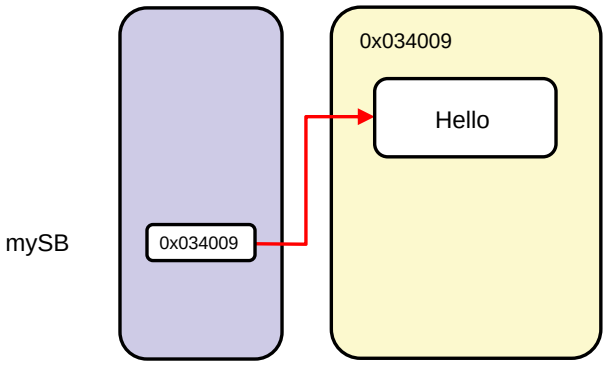
```
String myString = "Hello";
myString = myString + " World";
```



This slide offers a reminder of what happens when the strings `"Hello"` and `" World"` are concatenated. A new `String` object is created, and the reference for that object is assigned to `myString`.

StringBuilder: Declare and Instantiate

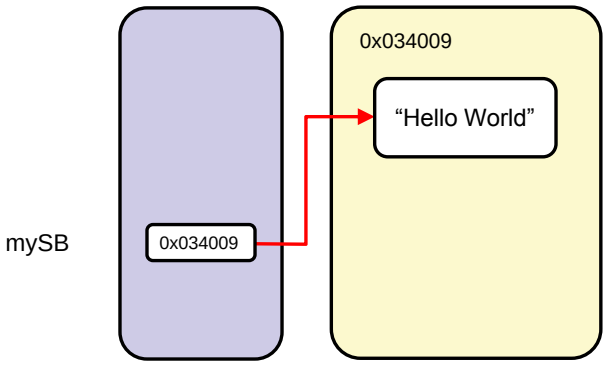
```
StringBuilder mySB = new StringBuilder("Hello");
```



This diagram shows the start of a sequence involving a `StringBuilder`. A new `StringBuilder` is instantiated, populated with the string `"Hello"`, and the reference for this new object is assigned to `mySB`.

StringBuilder Append

```
StringBuilder mySB = new StringBuilder("Hello");
mySB.append(" World");
```



To append the string `" World"`, all you need to do is call the `append` method and pass in "`World`". Note that no assignment (=) is necessary because there is already a reference to the `StringBuilder` object, and this `StringBuilder` object now contains a representation of the combined strings `"Hello World"`.

Even if you did assign the return type of the `append` method (which is `StringBuilder`), there would still be no object creation cost; the `append` method modifies the current object and returns the reference to that object, the one already contained in `mySB`.

Quiz

Which of the following statements are true? (Choose all that apply.)

a. The dot (`.`) operator creates a new object instance.

b. The `String` class provides you with the ability to store a sequence of characters.

c. The Java API specification contains documentation for all of the classes in a Java technology product.

d. `String` objects cannot be modified.

Answer: b, c, d

Exercise 7-2: Instantiate the `StringBuilder` object

1. Open the project **`Exercise_07-2`** or continue editing the previous exercise.
2. Instantiate a `StringBuilder` object (`sb`), initializing it to `firstName`, using the `StringBuilder` constructor.
3. Use the `append` method of the `StringBuilder` to append the last name back onto the first name. You can just use a String literal for the last name. Print the `StringBuilder` object and test your code. It should show the full name.
4. (Optional) Can you append the last name without using a String literal?

In this exercise, you instantiate a `StringBuilder` object, initializing it to `firstName` using the `StringBuilder` constructor.

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Primitive Data Types

- Integral types (`byte`, `short`, `int`, and `long`)
- Floating point types (`float` and `double`)
- Textual type (`char`)
- Logical type (`boolean`)

Many of the values in Java technology programs are stored as primitive data types. The slide lists the eight primitive types built in to the Java programming language. You have already learned about some of these and have been using them in your exercises and practices. Now you will see the remaining primitive types.

Some New Integral Primitive Types

Type	Length	Range
byte	8 bits	-2^7 to $2^7 - 1$ (–128 to 127, or 256 possible values)
short	16 bits	-2^{15} to $2^{15} - 1$ (–32,768 to 32,767, or 65,535 possible values)
int	32 bits	-2^{31} to $2^{31} - 1$ (–2,147,483,648 to 2,147,483,647, or 4,294,967,296 possible values)
long	64 bits	-2^{63} to $2^{63} - 1$ (–9,223,372,036854,775,808 to 9,223,372,036854,775,807, or 18,446,744,073,709,551,616 possible values)

There are four integral primitive types in the Java programming language. You have already been using the `int` data type, so the focus here is on the other three. Integral types are used to store numbers that do not have decimal portions. They are shown here in order of size.

- **`byte`:** If you need to store people's ages, a variable of type `byte` would work because `byte` types can accept values in that range.
- **`short`:** A `short` will hold 16 bits of data.
- **`long`:** When you specify a literal value for a `long` type, put a capital `L` to the right of the value to explicitly state that it is a `long` type. Integer literals are assumed by the compiler to be of type `int` unless you specify otherwise by using an `L` indicating `long` type.
- You can express any of the integral types as `binary` (0s and 1s). For instance, a `binary` expression of the number 2 is shown as an allowed value of the `byte` integral type. The binary value is `0b10`. Notice that this value starts with `0b` (that is, zero followed by either a lowercase or uppercase letter *B*). This indicates to the compiler that a `binary` value follows.

Examples of allowed literal values:

- `byte` = 2, -114, 0b10 (binary number)
- `short` = 2, -32699
- `int` (default type for integral literals) = 2, 147334778, 123_456_678
- `long` = 2, –2036854775808L, 1

Note: The only reason to use the `byte` and `short` types in programs is to save memory consumption. Because most modern desktop computers contain an abundance of memory, most desktop application programmers do not use `byte` and `short` types. This course uses primarily `int` and `long` types in the examples.

Floating Point Primitive Types

Type	Float Length
`float`	32 bits
`double` (default type for floating point literals)	64 bits

Example:

```
public float pi = 3.141592F;
```

There are two types for floating point numbers: `float` and `double`. Again, the focus is on the new data type here, the `float`. Floating point types are used to store numbers with values to the right of the decimal point, such as 12.24 or 3.14159.

- `float` is used to store smaller floating point numbers. A float variable can hold 32 bits.
- Floating point values are assumed to be of type `double` unless you specify by putting a capital `F` (`float`) to the right of the value to explicitly state that it is a `float` type, not a `double` type.

Examples of allowed literal values:

```
float = 99F, -327456,99.01F, 4.2E6F
```
(engineering notation for $4.2 * 10^6$)

```
double = -1111, 2.1E12, 99970132745699.999
```

Note: Use the `double` type when a greater range or higher accuracy is needed.

Textual Primitive Type

- The only primitive textual data type is `char`.
- It is used for a single character (16 bits).
- Example:
  - `public char colorCode = 'U';`

Single quotes must be used with char literal values.

Another data type that you use for storing and manipulating data is single-character information. The primitive type used for storing a single character (such as a '`y`') is `char`, which is 16 bits in size. The `Shirt` class shows the use of one textual literal value to specify the default value for a `colorCode`:

```
public char colorCode = 'U';
```

When you assign a literal value to a `char` variable, you must use single quotation marks around the character as shown in the code example above.

Java Language Trivia: Unicode

- Unicode is a standard character encoding system.
  - It uses a 16-bit character set.
  - It can store all the necessary characters from most languages.
  - Programs can be written so they display the correct language for most countries.

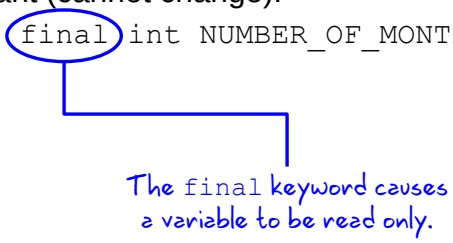
Character	UTF-16	UTF-8	UCS-2
A	0041	41	0041
c	0063	63	0063
Ö	00F6	C3 B6	00F6
垂	4E9C	E4 BA 9C	4E9C
𝄞	D834 DD1E	F0 9D 84 9E	N/A

Did You Know? Many older computer languages use American Standard Code for Information Interchange (ASCII), an 8-bit character set that has an entry for every English character, punctuation mark, number, and so on.

The Java programming language uses a 16-bit character set called Unicode that can store all the necessary displayable characters from the vast majority of languages used in the modern world. Therefore, your programs can be written so that they work correctly and display the correct language for most countries. Unicode contains a subset of ASCII (the first 128 characters).

Constants

- Variable (can change):
  - `double salesTax = 6.25;`
- Constant (cannot change):
  - `final int NUMBER_OF_MONTHS = 12;`



In this lesson, you have learned about variables that have values that you can change. In this section, you learn how to use constants to represent values that cannot change.

Assume that you are writing part of a scheduling application, and you need to refer to the number of months in a year. Make the variable a constant by using the `final` keyword to inform the compiler that you do not want the value of the variable to be changed after it has been initialized. Example:

```
final int NUMBER_OF_MONTHS = 12;
```

Any values that do not need to change are good candidates for a constant variable (for example, `MAX_COUNT`, or `PI`).

If someone attempts to change the value of a constant after it has already been assigned a value, the compiler gives an error message. If you modify your code to provide a different value for the constant, you need to recompile your program.

Guidelines for Naming Constants

You should name constants so that they can be easily identified. Generally, constants should be capitalized, with words separated by an underscore (_).

Quiz

The variable declaration `public int myInteger=10;` adheres to the variable declaration and initialization syntax.

a. True
b. False

Answer: a

Topics

- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Modulus Operator

Purpose	Operator	Example	Comments

Remainder	% (modulus)	`num1 = 31;`
		`num2 = 6;`
		
		`mod = num1 % num2;`
		
		`mod is 1`
			Remainder finds the remainder of the first number divided by the second number.
			5 R 1
			6 | 31
			30
			--- 	
			1
			Remainder always gives an answer with the same sign as the first operand.

Programs do a lot of mathematical calculating, from the simple to the complex. Arithmetic operators let you specify how the numerical values within variables should be evaluated or combined. The standard mathematical operators (often called *binary operators*) used in the Java programming language are shown in the tables in this section.

Note: The % is known as the modulus operator.

Combining Operators to Make Assignments

Purpose	Operator	Examples `int a = 6, b = 2;`	Result
Add to and assign	+=	a += b	a = 8
Subtract from and assign	-=	a -= b	a = 4
Multiply by and assign	*=	a *= b	a = 12
Divide by and assign	/=	a /= b	a = 3
Get remainder and assign	%=	a %= b	a = 0

Several very useful shortcuts are shown in the table above. You can combine any operator with the equal sign to abbreviate your code. For example:

```
a = a + b;
```

can be expressed as:

```
a += b;
```

More on Increment and Decrement Operators

Operator	Purpose	Example

`++`	Preincrement (`++variable`)	`int id = 6;`
		`int newId = ++id;`
		`id is 7, newId is 7`

	Postincrement (`variable++`)	`int id = 6;`
		`int newId = id++;`
		`id is 7, newId is 6`

`--`	Predecrement (`--variable`)	*(same principle applies)*
	Postdecrement (`variable--`)	

You have used increment and decrement operators before, placing them *after* the variable that you wish to affect. But did you know that these operators can come before (preincrement and predecrement) or after (postincrement and postdecrement) a variable.

When you put the `++` or `--` operator before a variable, the value is changed immediately. When you put the operator after the variable, it is not changed until after that expression is evaluated.

- In the first code example above, `id` is initialized to 6. In the next line, you see `newId = ++id`. Because the operator precedes `id`, this increment is immediately evaluated and, therefore, the value assigned to `newId` is 7.
- In the second code example, the `++` operator follows `id`, rather than precedes it. `id` was incremented after the assignment occurred. Therefore, `newId` is 6.
- These same behaviors apply to a decrement (`--`) operator, in regard to its placement before or after the variable.

Increment and Decrement Operators (++ and −−)

Examples:

```
1  int count=15;
2  int a, b, c, d;
3  a = count++;
4  b = count;
5  c = ++count;
6  d = count;
7  System.out.println(a + ", " + b + ", " + c + ", " + d);
```

Output:

```
15, 16, 17, 17
```

The example in the slide shows basic use of the increment and decrement operators:

```
int count=15;
int a, b, c, d;
a = count++;
b = count;
c = ++count;
d = count;
System.out.println(a + ", " + b + ", " + c + ", " + d);
```

The result of this code fragment is:

```
15, 16, 17, 17
```

Discussion: What is the result of the following code?

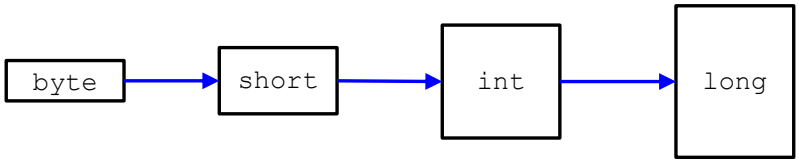
```
int i = 16;
System.out.println(++i + " " + i++ + " " + i);
```

Topics

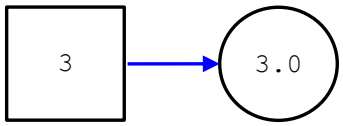
- Using the `String` class
- Using the Java API docs
- Using the `StringBuilder` class
- Doing more with primitive data types
- Using the remaining numeric operators
- Promoting and casting variables

Promotion

- Automatic promotions:
  - If you assign a smaller type to a larger type



  - If you assign an integral type to a floating point type



- Examples of automatic promotions:
  - `long intToLong = 6;`
  - `double intToDouble = 3;`

In some circumstances, the compiler changes the type of a variable to a type that supports a larger size value. This action is referred to as a *promotion*. Some promotions are done automatically by the compiler. These promotions include:

- If you assign a smaller type (on the right of the =) to a larger type (on the left of the =)
- If you assign an integral type to a floating point type (However, in some cases, such as an assignment of long to float, this could lead to loss of data.)

Caution with Promotion

Equation:

55555 * 66666 = 3703629630

Example of potential issue:

```
int num1 = 55555;
int num2 = 66666;
long num3;
num3 = num1 * num2;          //num3 is -591337666
```

Example of potential solution:

```
int num1 = 55555;
long num2 = 66666;
long num3;
num3 = num1 * num2;          //num3 is 3703629630
```

Changed from int to long

Before being assigned to a variable, the result of an equation is placed in a temporary location in memory. The location's size is always equal to the size of an `int` type or the size of the largest data type used in the expression or statement. For example, if your equation multiplies two `int` types, the container size will be an `int` type in size, or 32 bits.

If the two values that you multiply yield a value that is beyond the scope of an `int` type, (such as 55555 * 66666 = 3,703,629,630, which is too big to fit in an `int` type), the `int` value must be truncated to fit the result into the temporary location in memory. This calculation ultimately yields an incorrect answer because the variable for your answer receives a truncated value (regardless of the type used for your answer). To solve this problem, set at least one of the variables in your equation to the `long` type to ensure the largest possible temporary container size.

Caution with Promotion

Equation:

```
7 / 2 = 3.5
```

Example of potential issue:

```
1 int num1 = 7;
2 int num2 = 2;
3 double num3;
4 num3 = num1 / num2;          //num3 is 3.0
```

Example of potential solution:

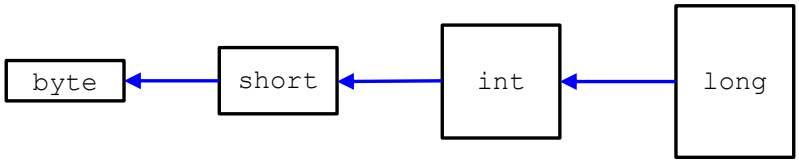
```
1 int num1 = 7;
2 double num2 = 2;  ——— Changed from int to double
3 double num3;
4 num3 = num1 / num2;          //num3 is 3.5
```

The same issue occurs with other data types. Before being assigned to a variable, the result of an equation is placed in a temporary location in memory. The location's size is always equal to the size of the largest data type used in the expression or statement. For example, if your equation divides two `int` types, the container size will be an `int` type in size, or 32 bits.

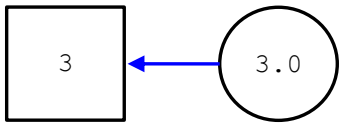
If the two values that you use yield a value that is beyond the scope of an `int` type, (such as 7 / 2 = 3.5), the value must be truncated to fit the result into the temporary location in memory. This calculation ultimately yields an incorrect answer because the variable for your answer receives a truncated value (regardless of the type used for your answer). To solve this problem, set at least one of the variables in your equation to the `double` type to ensure the largest possible temporary container size.

Type Casting

- When to cast:
  - If you assign a larger type to a smaller type



  - If you assign a floating point type to an integral type



- Examples of casting:
  - `int longToInt = (int)20L;`
  - `short doubleToShort = (short)3.0;`

Type casting lowers the range of a value, chopping it down to use a smaller amount of memory, by changing the type of the value (for example, by converting a `long` value to an `int` value). You do this so that you can use methods that accept only certain types as arguments, so that you can assign values to a variable of a smaller data type, or so that you can save memory.

The syntax for type casting a value is: **identifier = (target_type) value,** where:

- `identifier` is the name you assign to the variable
- `value` is the value you want to assign to the identifier
- `(target_type)` is the type to which you want to type cast the value. Notice that the `target_type` must be in parentheses.

Caution with Type Casting

Example of potential issue:

```
1 int myInt;
2 long myLong = 123987654321L;
3 myInt = (int) (myLong); // Number is "chopped"
4                         // myInt is -566397263
```

Safer example of casting:

```
1 int myInt;
2 long myLong = 99L;
3 myInt = (int) (myLong); // No data loss, only zeroes.
4                         // myInt is 99
```

The loss of precision with casting can sometimes lead to situations where numbers are truncated, leading to errors in calculations.

Caution with Type Casting

- Be aware of the possibility of lost precision.

Example of potential issue:

```
1 int myInt;
2 double myPercent = 51.9;
3 myInt = (int) (myPercent); // Number is "chopped"
4                            // myInt is 51
```

If you type cast a `float` or `double` value with a fractional part to an integral type such as an `int`, all decimal values are lost. However, this method of type casting is sometimes useful if you want to truncate the number down to the whole number (for example, 51.9 becomes 51).

Using Promotion and Casting

Example of potential issue:

```
1 int num1 = 53; // 32 bits of memory to hold the value
2 int num2 = 47; // 32 bits of memory to hold the value
3 byte num3;     // 8 bits of memory reserved
4 num3 = (num1 + num2); // causes compiler error
```

Solution using a larger type for num3:

```
1 int num1 = 53;
2 int num2 = 47;
3 int num3;      ——— Changed from byte to int
4 num3 = (num1 + num2);
```

Solution using casting:

```
1 int num1 = 53; // 32 bits of memory to hold the value
2 int num2 = 47; // 32 bits of memory to hold the value
3 byte num3;     // 8 bits of memory reserved
4 num3 = (byte)(num1 + num2); // no data loss
```

Assigning a variable or an expression to another variable can lead to a mismatch between the data types of the calculation and the storage location that you are using to save the result. Specifically, the compiler will either recognize that precision will be lost and not allow you to compile the program, or the result will be incorrect. To fix this problem, variable types have to be either promoted to a larger size type, or type cast to a smaller size type. In the above example, the compiler assumes that because you are adding `int` values, the result will overflow the space allocated for a `byte`.

A `byte`, though smaller than an `int`, is large enough to store a value of `100`. However, the compiler will not make this assignment and, instead, issues a “possible loss of precision” error because a `byte` value is smaller than an `int` value. To fix this problem, you can either type cast the right-side data type down to match the left-side data type, or declare the variable on the left side (`num3`) to be a larger data type, such as an `int`.

Compiler Assumptions for Integral and Floating Point Data Types

- Most operations result in an `int` or `long`:
  - `byte`, `char`, and `short` values are automatically promoted to `int` prior to an operation.
  - If an expression contains a `long`, the entire expression is promoted to `long`.
- If an expression contains a floating point, the entire expression is promoted to a floating point.
- All literal floating point values are viewed as `double`.

The Java technology compiler makes certain assumptions when it evaluates expressions. You must understand these assumptions to make the appropriate type casts or other accommodations. The next few slides give examples.

Automatic Promotion

Example of potential problem:

```
short a, b, c;
a = 1 ;
b = 2 ;
c = a + b ; //compiler error
```

a and b are automatically promoted to integers.

Example of potential solutions:

- Declare `c` as an `int` type in the original declaration:

```
int c;
```

- Type cast the `(a+b)` result in the assignment line:

```
c = (short)(a+b);
```

In the following example, an error occurs because two of the three operands (`a` and `b`) are automatically promoted from a `short` type to an `int` type before they are added. In the last line, the values of `a` and `b` are converted to `int` types and the converted values are added to give an `int` result. Then the assignment operator (=) attempts to assign the `int` result to the `short` variable (`c`). However, this assignment is illegal and causes a compiler error.

The code works if you do either of the following:

- Declare `c` as an `int` in the original declaration:

```
int c;
```

- Type cast the (a+b) result in the assignment line:

```
c = (short)(a+b);
```

Using a `long`

```
1 public class Person {
2
3   public int ageYears = 32;
4
4   public void calculateAge() {
5
6     int ageDays = ageYears * 365;
7     long ageSeconds = ageYears * 365 * 24L * 60 * 60;
8
9     System.out.println("You are " + ageDays + " days old.");
10    System.out.println("You are " + ageSeconds + " seconds old.");
11
12   } // end of calculateAge method
13  } // end of class
```

Using the L to indicate a long will result in the compiler recognizing the total result as a long.

The code example uses principles from this section to calculate a person's age in days and seconds. Because the `ageSeconds` variable is declared as a `long`, one of the literal values used as operands in the assigned expression must be initialized as a `long` value ('L') so that the compiler will allow the assignment.

Using Floating Points

Example of potential problem:

Expressions are automatically promoted to floating points.

```
int num1 = 1 + 2 + 3 + 4.0;                    //compiler error
int num2 = (1 + 2 + 3 + 4) * 1.0;              //compiler error
```

Example of potential solutions:

- Declare `num1` and `num2` as `double` types:

```
double num1 = 1 + 2 + 3 + 4.0;                 //10.0
double num2 = (1 + 2 + 3 + 4) * 1.0;           //10.0
```

- Type cast `num1` and `num2` as `int` types in the assignment line:

```
int num1 = (int)(1 + 2 + 3 + 4.0);             //10
int num2 = (int)((1 + 2 + 3 + 4) * 1.0);       //10
```

If an expression contains a floating point, the entire expression is promoted to a floating point.

Floating Point Data Types and Assignment

- Example of potential problem:

```
float float1 = 27.9;  //compiler error
```

- Example of potential solutions:
  - The `F` notifies the compiler that 27.9 is a `float` value:

```
float float1 = 27.9F;
```

  - `27.9` is cast to a `float` type:

```
float float1 = (float) 27.9;
```

Just as integral types default to `int` under some circumstances, values assigned to floating point types always default to a `double` type, unless you specifically state that the value is a `float` type.

For example, the following line causes a compiler error. Because `27.9` is assumed to be a `double` type, a compiler error occurs because a `double` type value cannot fit into a `float` variable.

```
float float1 = 27.9;   //compiler error
```

Both of the following work correctly:

- The `F` notifies the compiler that `27.9` is a `float` value:

```
float float1 = 27.9F;
```

- `27.9` is cast to a `float` type:

```
float float1 = (float) 27.9;
```

Quiz

Which statements are true?

a. There are eight primitive types built in to the Java programming language.

b. `byte`, `short`, `char`, and `long` are the four integral primitive data types in the Java programming language.

c. A `boolean` type variable holds `true`, `false`, and `nil`.

d. `short Long = 10;` is a valid statement that adheres to the variable declaration and initialization syntax.

Answer: a, d

- a is correct.
- b is incorrect. It should be `byte`, `short`, `int`, and `long`.
- c is incorrect because a `boolean` type variable holds only `true` and `false`.
- d is correct. `long` is a reserved keyword but `Long` is not.

Exercise 7-3: Declare a `Long`, `Float`, and `Char`

1. Open the project **`Practice_07-3`** in NetBeans.
2. Declare a `long`, using the `L` to indicate a long value. Make it a very large number (in the billions).
3. Declare and initialize a `float` and a `char`
4. Print the `long` variable with a suitable label.
5. Assign the `long` to the `int` variable. Correct the syntax error by casting the `long` as an `int`.
6. Print the `int` variable. Note the change in value when you run it.

In this exercise, you experiment with the data types introduced in this lesson. You:

Declare and initialize variables

Cast one numeric type to another

Summary

In this lesson, you should have learned how to:

- Describe the `String` class and use some of the methods of the `String` class
- Use the JDK API documentation to search for and learn how to use a class
- Use the `StringBuilder` class to manipulate string data
- Create a constant by using the final keyword in the variable declaration
- Describe how the Java compiler can use promotion or casting to interpret expressions and avoid a compiler error

Practices Overview

- 7-1: Manipulating Text