Revised: 12/28/2020

Object-Oriented Programming I

Michael C. Hackett

Assistant Professor, Computer Science

Community
College
of Philadelphia

Lecture Topics

- Objects and Classes
- Primitive Data Types
- Fields/Variables
- The main method
- Creating an Instance of an Object
  - Using an Object’s Fields
  - Using Standard Output
  - Creating Multiple Instances of an Object
- Strings
- Methods
- Using Standard Input
- Notes:
  - Java Variables
  - Java Literals
  - Java Strings
  - Escape Sequences
  - Parsing Numbers from Strings
  - Python and Java Syntax/Statement Comparisons

What is Object-Oriented Programming?

- A programming paradigm where software is written so that a program functions as a system of objects.
    - A different way of designing software as opposed to the script-like, procedural model of how we wrote Python programs.

- The objects interact with each other to complete the program’s tasks.

- Software objects contain information about themselves and allow interaction with other objects.

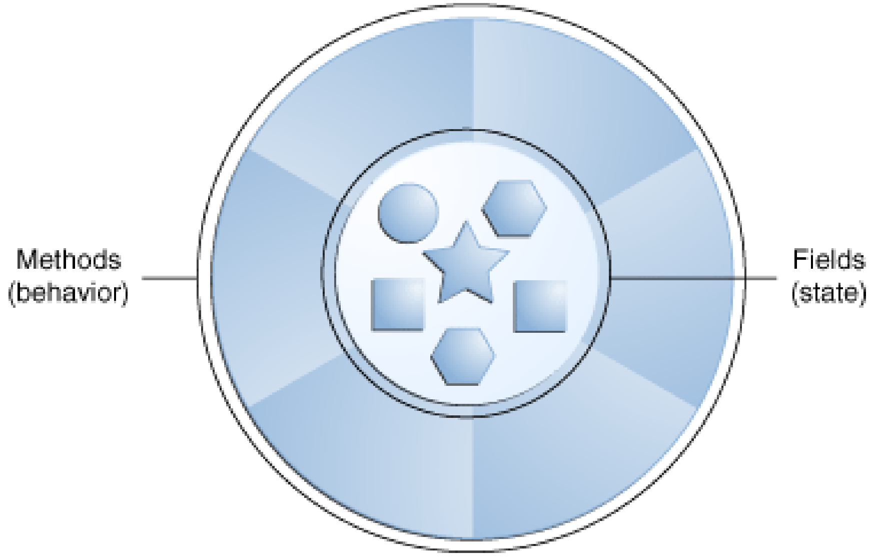
What are Objects?

- A software object is *conceptually* similar to a real-world object.
- Real world objects all have two features:
  - They have ***attributes***- properties that make the object unique.
    - A bicycle's attributes could be its current speed, color, tire size, etc.
  - They have ***behaviors***- actions that the object can do.
    - A bicycle's behaviors could be pedaling, braking, turning left or right, changing gear, etc.

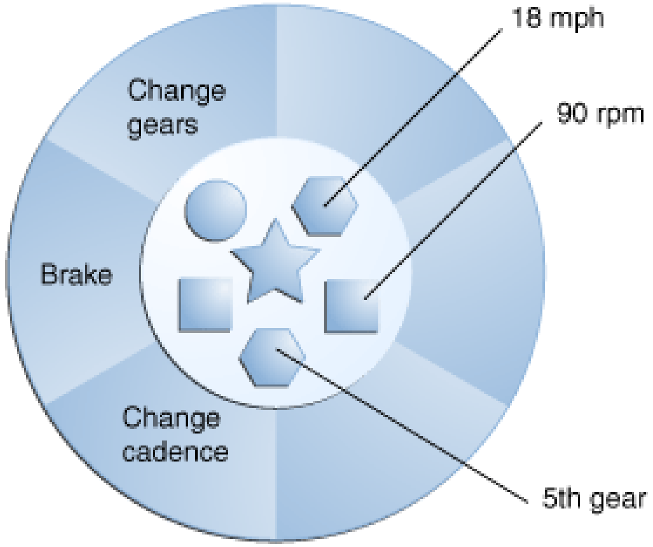
- When we model a software object, it too has attributes and behaviors.
  - Objects store their attributes in variables referred to as ***fields.***
  - Objects expose their behaviors as ***methods.***

What are Objects?

Software Object



Bicycle modeled as an Object



The Four Pillars of Object-Oriented Programming

- **Abstraction**
  - The inner workings of the object (it’s code, logic, etc.) are contained within the object.
  - We only interact with the object through the behaviors it has defined for us.

- **Encapsulation**
  - The object contains its own data.
  - The object dictates how and if this data is accessible.

The Four Pillars of Object-Oriented Programming

- **Inheritance**
  - *Introduced in CSCI 112*
  - Objects can have parent-child relationships.
  - Child objects inherit certain attributes and behaviors from its parent object.

- **Polymorphism**
  - *Introduced in CSCI 112*
  - Objects can be treated as more than one type of object
  - Adds layers of flexibility to a program

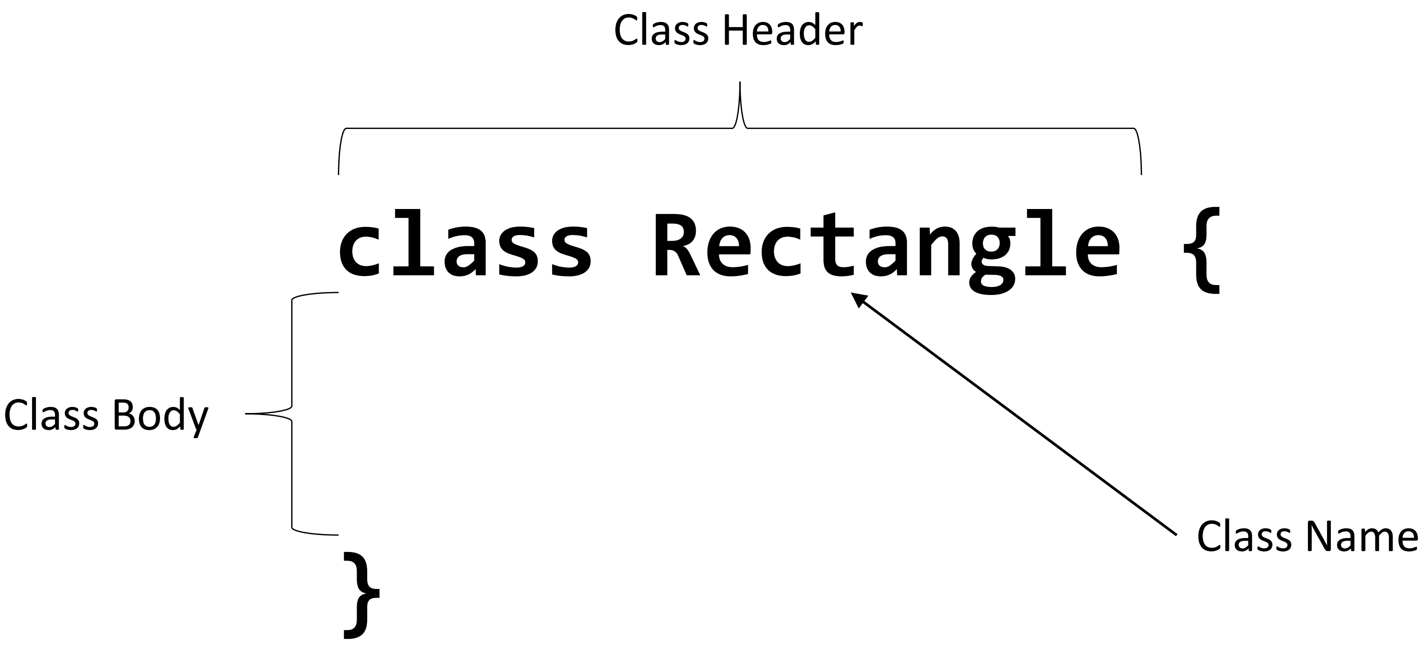
Class Declaration

- Objects are defined using a **class**.
  - The class contains the object's fields and methods.
- Classes are declared using the class keyword followed by the desired name of the class.
  - Braces identify the beginning and end of the **class body**

```
class Rectangle {


}
```

Class Declaration



Class Header

```
class Rectangle {

}
```

Class Body

Class Name

Primitive Data Types

- Java has eight primitive data types
  - Think of the standard data types (like int, float, bool, and str) from Python
- Four Integer Types
  - **`byte`**, **`short`**, **`int`**, **`long`**
- Two Floating-Point Types
  - **`float`**, **`double`**
- One Boolean Type
  - **`boolean`**
- One Character Type
  - **`char`**

Primitive Data Types

- Integer Types
  - **`byte`** (8 bits)
    - Can represent any integer between -128 and 127
  - **`short`** (16 bits)
    - Can represent any integer between -32,768 and 32,767
  - **`int`** (32 bits)
    - Can represent any integer between -2,147,483,648 and 2,147,483,647
    - Most frequently used integer primitive.
  - **`long`** (64 bits)
    - Can represent any integer between -2^{63} and $2^{63}-1$

Primitive Data Types

- Floating-Point Types
  - **`float`** (32 bits)
    - Can represent values between ~ ±3.4×10^{38} with 7 significant digits.
  - **`double`** (64 bits)
    - Can represent values between ~ ±1.7×10^{308} with 15 significant digits.

- Boolean Type
  - **`boolean`** (1 bits)
    - Can be **`true`** or **`false`**.
    - Used for decision making.
  - Depending on the OS's memory management, it may not be able to allocate a single bit of memory.
    - The OS may allocate an entire byte (8 bits), though only one of the bits will be used.

Primitive Data Types

- **`char`** (16 bits)
  - Can represent a single, 16-bit Unicode character.
  - UTF-16 character table for reference:

    http://www.fileformat.info/info/charset/UTF-16/list.htm

- Together, multiple chars make up a **`String`** object.
  - Strings are not primitive types in Java (they are *objects*), but they are functionally the same as Strings seen in Python.
  - Because they are used extensively, Strings will be taught alongside primitive types in this lecture

Fields

- The fields and methods (*attributes and behaviors*) of a class are referred to as **class members**

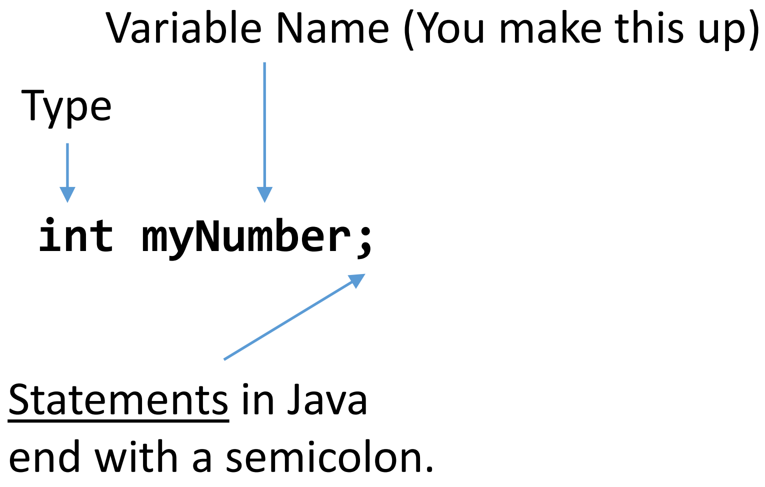
- A ***field*** is simply a variable that stores data for the class.

- When declaring variables in Java, we must specify the variable’s type.
  - Unlike Python, which uses dynamic typing (any data type can be assigned to any variable at any time), Java uses static typing which means that each variable may only reference one type of data or object.

Fields

- When a variable is declared in Java, you are stating:
  - The type of data this variable will reference.
  - The name of the variable.
- Examples:

```
int age;
double temperature;
char letter;
boolean ready;
```



Variable Name (You make this up)

Type

```
int myNumber;
```

Statements in Java
end with a semicolon.

- Remember, variables in Java are statically typed.
  - It cannot reference values of a different data type after it has been declared.

Fields

- Names must start with a letter, dollar sign, or underscore.
- Names can contain numbers but **cannot** start with numbers.
- Aside from letters, dollar signs, underscores, and numbers, no other characters may be used.
- Names cannot contain spaces. Use underscores, if necessary.

`int someName;`	**Valid.**
`int _someName;`	**Valid. Can start with underscore.**
`int some_Name;`	**Valid. Can contain any underscores.**
`int $someName;`	**Valid. Can start with dollar sign.**
`int some$Name;`	**Valid. Can contain any dollar signs.**
`int 0someName;`	**INVALID. Can't start with a number.**
`int some0Name;`	**Valid. Can contain any numbers.**
`int some Name;`	**INVALID.**
`int some_Name;`	**Valid.**

Fields

```
class Rectangle {

   int width;
   int length;

}
```

- This Rectangle class now contains two fields called length and width, both of which may only reference int data.
- Java does not require indentation like Python does.
  - Indentation does make the class's code neater and easier to read.

Fields

- Variable names in Java normally start with a lowercase letter.
  - Class names normally start with an uppercase letter.
  - Hence, we named the class **`Rectangle`** and the fields **`length`** and **`width`**

- "Camel-case" is the convention used for variable names in Java.
  - For variable names that are multiple words long, the first letter of every subsequent word should be capitalized.

```
bottlesOfBeerOnTheWall = 99
hasBeenDeleted = True
```

The main method

- The main method is the starting point for a Java application.
  - The main method must be written as shown below.
    - We’ll ignore much of the keywords like “public” and “static” for the time being, as those concepts will be covered in later lectures.

```
class TestProgram {

  public static void main(String[] args) {

  }

}
```

main method

Creating an Instance of an Object

- In a second class named TestProgram, we will declare a variable of the Rectangle type in the main method (shown below).
    - Note that declaring a variable within a method (called a **local variable** as opposed to a **class variable**/field) is the same as how we declared the fields int the Rectangle class.

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
    }
}
```

Declares a variable named demo which will only be able to reference a Rectangle object

Creating an Instance of an Object

- In order to use a Rectangle object, the object must be instantiated.
  - ***Instantiation*** is the term used when you create an ***instance*** of an object from a class.
- In the example below, a new instance of a Rectangle object is constructed and is assigned to the "demo" variable.

```
class TestProgram {

   public static void main(String[] args) {
      Rectangle demo;
      demo = new Rectangle();
   }
}
```

Instantiates a new Rectangle object and assigns it to the demo variable

Creating an Instance of an Object

- Objects are instantiated using a special method called a ***constructor***.
    - Constructors are used to “set up” or *construct* an instance of an object.
- When there are no constructors present in a class, the compiler automatically adds a ***default constructor***.
    - This guarantees every object has a constructor.
    - Constructors are covered in a later lecture.

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();
    }
}
```

Using an Object’s Fields

- We can access an object’s fields using dot notation.

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();

        demo.width = 3;
        demo.length = 7;
    }
}
```

Assigns 3 to the width field and 7 to the length field

Using Standard Output

- The **`System`** object is provided by Java and allows access to standard input and output.
  - The standard output stream is simple text output to the console/terminal.
  - The standard input stream is the keyboard (which we will later use to allow a user to type input into a program.)

- The **`System`** object has a field named “out”, which is a **`PrintStream`*** object.
  - This object handles all output to the standard output stream, ultimately printing data to the terminal/console.

* You don’t have to worry about the technical specifics of a PrintStream object.

Using Standard Output

- There are two methods we will be accessing from **`System`**'s **`out`** field
  - **`print`** and **`println`**

- Like accessing an object's fields, an object's methods are also accessed ("*called*") using dot notation:
  - **`System.out.print()`**
  - **`System.out.println()`**
  - The above two statements call the **`print`** or **`println`** methods from the **`System`** object's **`out`** field.

- The parentheses after a method name is for the parameter list.
  - Any data passed as a parameter to the **`print`** or **`println`** methods will be printed to the console/terminal.

Using Standard Output

- After printing the supplied information, the **println** method will advance to the next line of output.

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();

        demo.width = 3;
        demo.length = 7;
        System.out.println(demo.width);
        System.out.println(demo.length);
    }
}
```

Output:

```
3
7
```

Using Standard Output

- After printing the supplied information, the **`print`** method will **NOT** advance to the next line of output.

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();

        demo.width = 3;
        demo.length = 7;
        System.out.print(demo.width);
        System.out.print(demo.length);
    }
}
```

Output:

```
37
```

Using Standard Output

- Both methods only accept **one** argument.
- We could NOT do the following:

Compile-Time Error.

```
demo.width = 3;
demo.length = 7;
System.out.print(demo.width, demo.length);
```

Using an Object’s Fields

- This example demonstrates how the object’s fields can be updated:

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo;
        demo = new Rectangle();

        demo.width = 3;
        demo.length = 7;
        System.out.println(demo.width);
        System.out.println(demo.length);
        demo.length = 18;
        System.out.println(demo.width);
        System.out.println(demo.length);
    }
}
```

Output:

```
3
7
3
18
```

Creating Multiple Instances of an Object

- A major benefit of object-oriented programming and classes is the ability to reuse parts of our program.

- For example, we can create multiple instances of Rectangle objects, each with their own unique lengths and widths.

- Since they all draw from the same class, all Rectangle objects will have the same types attributes and the same behaviors

Creating Multiple Instances of an Object

- A quick note:
  - You can declare and initialize a variable on the same line.

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo = new Rectangle();
    }
}
```

Creating Multiple Instances of an Object

- Four distinct Rectangle objects are instantiated:

```
class TestProgram {

   public static void main(String[] args) {
      Rectangle demo = new Rectangle();
      Rectangle demo2 = new Rectangle();
      Rectangle demo3 = new Rectangle();
      Rectangle demo4 = new Rectangle();
   }
}
```

Creating Multiple Instances of an Object

- The four Rectangle objects can each have different values assigned to their width and length fields (only using the width field in this example):

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo = new Rectangle();
        Rectangle demo2 = new Rectangle();
        Rectangle demo3 = new Rectangle();
        Rectangle demo4 = new Rectangle();
        demo.width = 4;
        demo2.width = 7;
        demo3.width = 2;
        demo4.width = 8;
    }
}
```

Creating Multiple Instances of an Object

- Printing the values of the width fields of these objects shows they each store a unique value in their width fields
- The takeaway here is we easily created four objects that have the same exact abilities (*abstraction*) and maintain their own data (*encapsulation*)

```
class TestProgram {

    public static void main(String[] args) {
        Rectangle demo = new Rectangle();
        Rectangle demo2 = new Rectangle();
        Rectangle demo3 = new Rectangle();
        Rectangle demo4 = new Rectangle();
        demo.width = 4;
        demo2.width = 7;
        demo3.width = 2;
        demo4.width = 8;
        System.out.println(demo.width);
        System.out.println(demo2.width);
        System.out.println(demo3.width);
        System.out.println(demo4.width);
    }
}
```

Output:

```
4
7
2
8
```

Strings

- A ***String*** is an object that contains a sequence of characters.
  - It contains a sequence of **char** data

- The sequence of characters in a String can include any number of letters, numbers, symbols, spaces

Strings

```
class Circle {

   double radius;
   String color;

}
```

- This Circle class contains two fields called radius (double type) and color (String type).
- String is an object type, hence the uppercase S, per naming conventions

Strings

```
class Circle {

   double radius;
   String color;

}
```

- When variables are declared but not initialized with data (like the fields above)…
  - **byte**, **short**, **int**, and **long** variables will have an initial value of **0**
  - **float** and **double** variables will have an initial value of **0.0**
  - **boolean** variables will have an initial value of **false**
  - **char** variables will have an initial value of **' '** (a blank space)
  - Object-type variables will be **null** – meaning the variable does not reference anywhere in memory

Strings

- This example displays the initial values of a Circle object’s fields:

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();

        System.out.println(demo.radius);
        System.out.println(demo.color);
    }
}
```

Output:

```
0
null
```

Strings

- This example assigns values to a Circle object’s fields and displays them
- It demonstrates using a String literal
  - No big difference from Python, except you can only use double quotes to indicate the beginning and end of a String literal in Java

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        demo.radius = 40;
        demo.color = "Blue";

        System.out.println(demo.radius);
        System.out.println(demo.color);
    }
}
```

Output:

```
40
Blue
```

Methods

- A ***subroutine*** is a group of self-contained instructions that are executed when called.
  - A **function** is a subroutine that returns data when called.

    ```
    answer = Math.sqrt(x);
    ```

  - A **procedure** is a subroutine that does not return data when called.

    ```
    System.out.println("Hello World!");
    ```

  - A **method** is a subroutine (function or procedure) that is part of a software object.

    ```
    someString.toLowerCase();
    ```

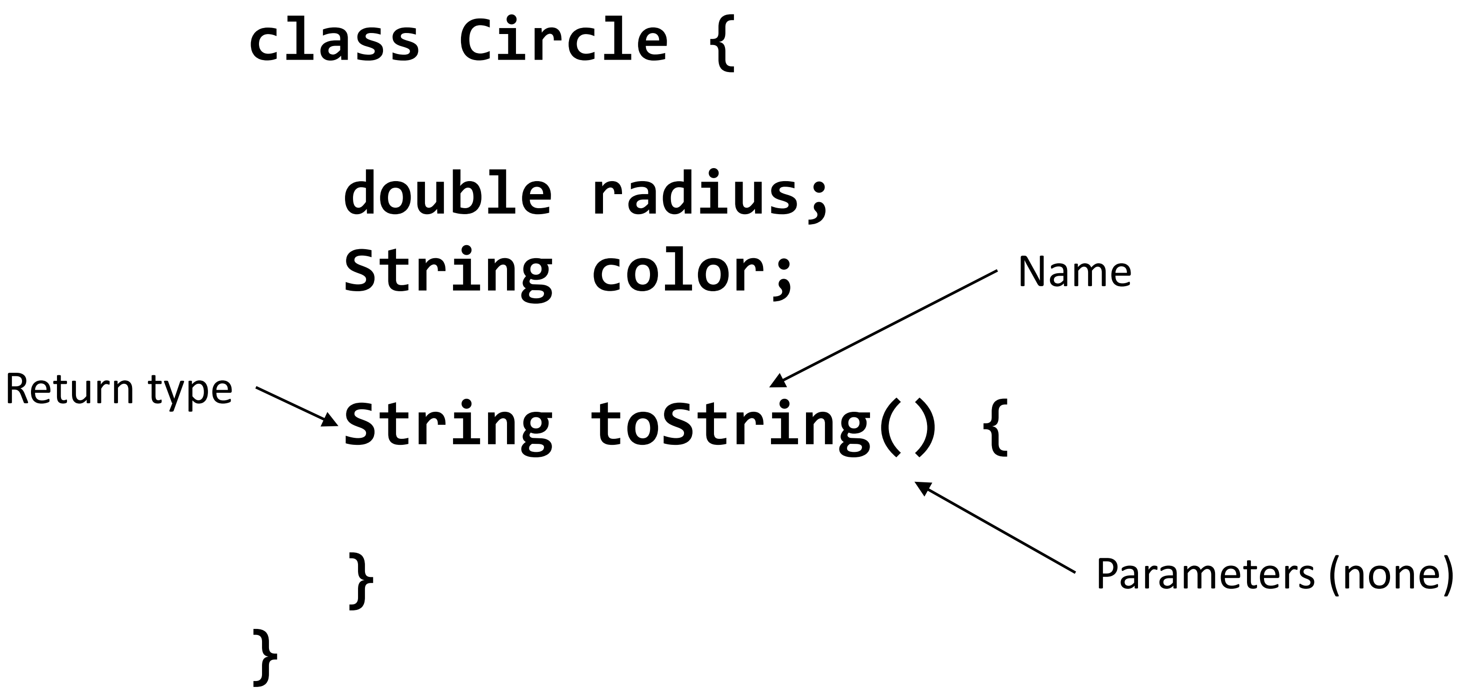
- All four terms are generally used interchangeably, though these are the correct definitions.

Methods

- The first example of defining a method in a class will be to prepare and return a descriptive String about the Circle object.
  - The String will contain (in its text) the current values of the radius and color fields.

- At a minimum, methods must specify
  - **The type of data they return**
    - This method will return a String
  - **The name of the method**
    - We will name this method “toString”
  - **A parameter list**
    - This method will have no parameters
    - We’ll see the use of parameters in the next lecture

Methods



```
class Circle {

   double radius;
   String color;

   String toString() {

   }
}
```

Name

Return type

Parameters (none)

Methods

- Let’s have this method return a String that is in the following format:

  `Circle radius: X and color: Y`

- Where X is replaced with the current value of the radius field and Y is replaced with the current value of the color field

- This will involve concatenation, with no major differences from concatenation in Python
  - In Java, though, numeric data can be concatenated with Strings without the need for typecasting.

Methods

```
class Circle {

   double radius;
   String color;

   String toString() {
      String text = "Circle radius: " + radius + " and color: " + color;
   }
}
```

Methods

```
class Circle {

    double radius;
    String color;

    String toString() {
        String text = "Circle radius: " +
                      radius +
                      " and color: " +
                      color;

    }
}
```

Alternative formatting for long concatenations

Methods

- Finally, we will need to explicitly state when (and what) to return from the method using a **return statement**.
- Be sure the type of data returned matches the return type that is specified in the method's declaration.

```
class Circle {

    double radius;
    String color;

    String toString() {
        String text = "Circle radius: " +
                       radius +
                       " and color: " +
                       color;
        return text;
    }
}
```

Methods

- This example calls the Circle object’s toString method, which stored the returned value to a variable named “description”
- The String referenced by “description” is printed

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        demo.radius = 40;
        demo.color = "Blue";

        String description = demo.toString();
        System.out.println(description);
    }
}
```

Output:

```
Circle radius: 40 and color: Blue
```

Methods

- This example essentially does the same thing, but it calls the object's toString method *within* the call to the println method.

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        demo.radius = 40;
        demo.color = "Blue";

        System.out.println(demo.toString());
    }
}
```

Output:

```
Circle radius: 40 and color: Blue
```

Methods

```
class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Circle demo2 = new Circle();

        demo.radius = 40;
        demo.color = "Blue";

        demo2.radius = 25;
        demo2.color = "Green";

        System.out.println(demo.toString());
        System.out.println(demo2.toString());

    }
}
```

Output:

```
Circle radius: 40 and color: Blue
Circle radius: 25 and color: Green
```

Using Standard Input

- An easy way to get keyboard entries is using a Scanner object.
- Unlike the System and String objects, the Scanners object needs to be imported.
- Include the following line at the very beginning of your source code, before the class header, in the file where the Scanner object is needed.

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

Using Standard Input

- Scanners, like the Rectangle and Circle objects we created, will need to be instantiated.
- We provide this new instance of a Scanner with System.in
  - System.in references the standard input stream, much like how System.out references the standard output stream.

```
import java.util.Scanner;

class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Scanner keyboard = new Scanner(System.in);

    }
}
```

When using a Scanner to get keyboard input, I usually name it “keyboard.” You can name it whatever you like, though.

Using Standard Input

- We'll prompt the user to enter the color of the Circle.
  - We will assign the entered value to the appropriate field in a Circle object.

```
import java.util.Scanner;

class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Scanner keyboard = new Scanner(System.in);
        demo.radius = 16;
        System.out.print("Enter the circle's color: ");

    }
}
```

Using Standard Input

- The Scanner’s nextLine method returns the user’s input as String

```
import java.util.Scanner;

class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Scanner keyboard = new Scanner(System.in);
        demo.radius = 16;
        System.out.print("Enter the circle's color: ");
        String userColor = keyboard.nextLine();

    }
}
```

Using Standard Input

```
import java.util.Scanner;

class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Scanner keyboard = new Scanner(System.in);
        demo.radius = 16;
        System.out.print("Enter the circle's color: ");
        String userColor = keyboard.nextLine();

        demo.color = userColor;
        System.out.println(demo.toString());

    }
}
```

Output:

```
Enter the circle’s color: Orange
Circle radius: 16 and color: Orange
```

Using Standard Input

- Since the radius field is numeric, any String returned by the Scanner’s nextLine method will need to be converted to a numeric type.

- Convert a String to an int:

```
    int x = Integer.parseInt(stringToConvert);
```

- Convert a String to a double:

```
    double y = Double.parseDouble(stringToConvert);
```

Using Standard Input

```
import java.util.Scanner;

class TestProgram {

    public static void main(String[] args) {
        Circle demo = new Circle();
        Scanner keyboard = new Scanner(System.in);

        System.out.print("Enter the circle's color: ");
        String userColor = keyboard.nextLine();
        demo.color = userColor;

        System.out.print("Enter the circle's radius: ");
        String userRadius = keyboard.nextLine();
        demo.radius = Integer.parseInt(userRadius);

        System.out.println(demo.toString());
    }
}
```

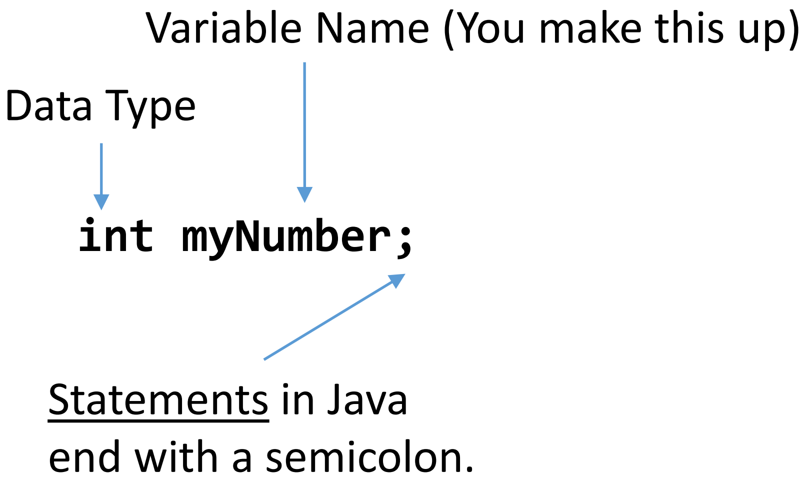
Output:

```
Enter the circle’s color: Orange
Enter the circle’s radius: 21
Circle radius: 21 and color: Orange
```

Notes on Java Variables

- When a variable is ***declared*** in Java, you are stating:
  - The type of data this variable will reference.
  - The name of the variable.
- Examples:

```
int age;
double temperature;
char letter;
boolean ready;
```



Variable Name (You make this up)

Data Type

```
int myNumber;
```

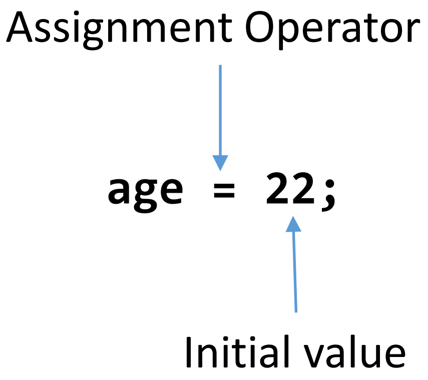
Statements in Java end with a semicolon.

- Variables in Java are ***statically typed***.
  - It cannot reference values of a different data type after it has been declared.

Notes on Java Variables

- A variable becomes ***initialized*** when an initial value is assigned to the variable.
- Assignment operator: **=**
  - Same as Python
- Examples (all previously declared in the last slide):

```
age = 22;
temperature = 80.5;
letter = 'A';
ready = true;
```



Assignment Operator

```
age = 22;
```

Initial value

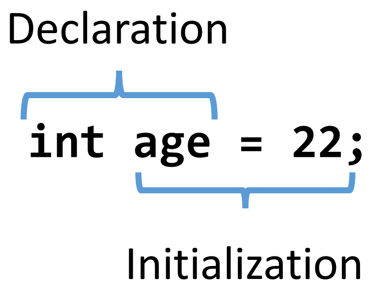
- Note: You cannot initialize a variable that has not yet been declared!

Notes on Java Variables

- You can assign an initial value when you declare a new variable.

- Examples:

```
int age = 22;
double temperature = 80.5;
char letter = 'A';
boolean ready = true;
```


Declaration

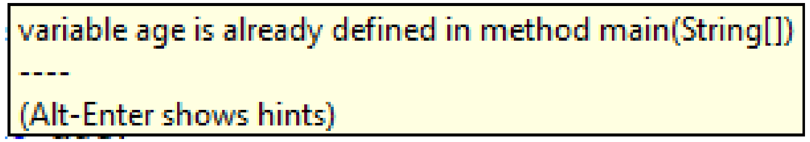
```
int age = 22;
```

Initialization

Notes on Java Variables

- Variable names must be unique, regardless of data type.

```
int age;
double age;
```



variable age is already defined in method main(String[])

(Alt-Enter shows hints)

Compile-Time Error will occur.

- A compile-time error occurs when you compile your code.
- A run-time error occurs when your program is running.

- A variable's data type cannot be changed after declaration.
- The names of the primitive data types are all Java keywords.
  - Keywords cannot be the name of a variable.

```
int boolean;
```

Compile-Time Error.

Notes on Java Literals

- byte, short and int literals can be expressed in
  - Decimal (Base 10)
  - Octal (Base 8), or
  - Hexadecimal (Base 16)

- Base 10 Literal (No prefix): `int decimalNumber = 100;`
- Base 8 Literal* (0 prefix): `int octalNumber = 0144;`
- Base 16 Literal* (0x prefix): `int hexNumber = 0x64;`

*For the purpose of this course, we will only be using base 10 values.
It’s good to know that other numeric literals exist, though.

Notes on Java Literals

- long literals (Must add a lowercase or capital L to the end):

```
long exampleLongLiteral = 255L;
```

- A capital L is preferred, as lowercase L may be mistaken for a 1.
  - **`255l`** vs **`2551`** vs **`255L`**

Notes on Java Literals

- double literals (Nothing special needs to be done):

```
double exampleDoubleLiteral = 255.23;
```

- float literals (Must add upper- or lowercase f to the end):

```
float myExampleFloat = 15.5f;
```

The compiler (like with many other programming languages/compilers) interprets literal fractional numbers as doubles by default. To differentiate float literals from double literals, float literals must end with a lowercase f.

Notes on Java Literals

- char literals can be expressed as a character literal, a Unicode literal or a number.
- **Must be in single quotes!!**

```
char exampleCharLiteral = 'A';
char exampleCharUnicodeLiteral = '\u0041';
char exampleCharDecimalLiteral = 65;
```

Any classwork will use character literals like exampleCharLiteral above.

Notes on Java Strings

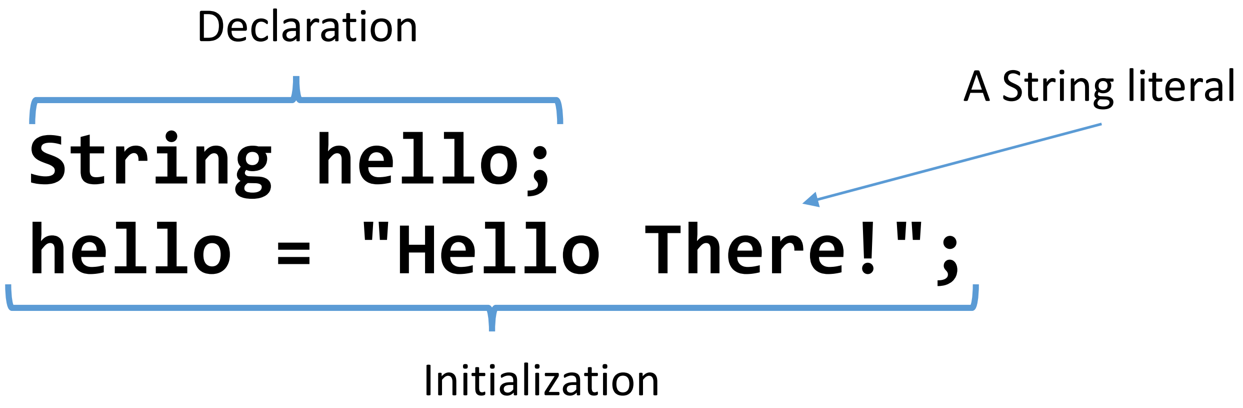
- Strings, like primitives, are declared:
  - First stating the data type.
  - Then stating the variable name.

```
String hello;
```

Capital S

Notes on Java Strings

- Use the assignment operator: **=**
- A ***String literal*** is any source code representation of a sequence of characters in double quotation marks.



Declaration

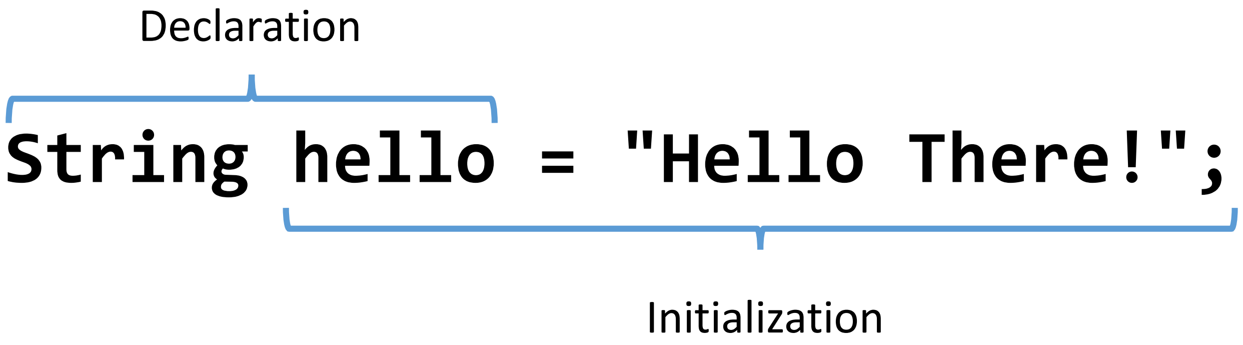
A String literal

```
String hello;
hello = "Hello There!";
```

Initialization

Notes on Java Strings

- Declaration and initialization can be done in one statement.



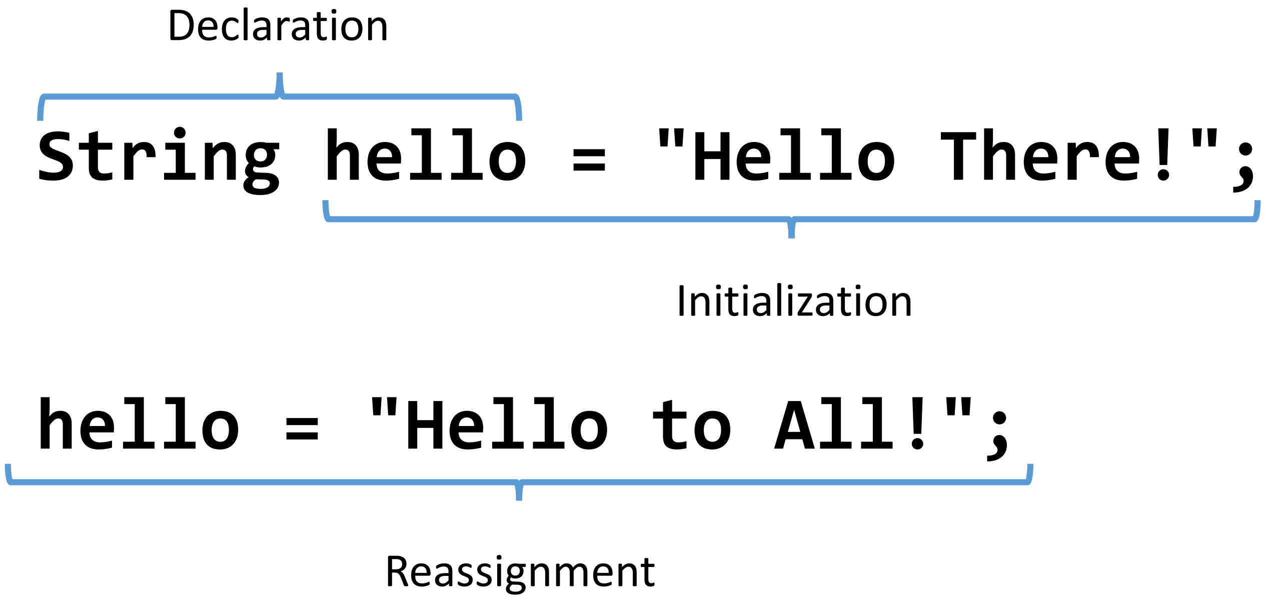
Declaration

```
String hello = "Hello There!";
```

Initialization

Notes on Java Strings

- Use the assignment operator when re-assigning: **=**



Declaration

```
String hello = "Hello There!";
```

Initialization

```
hello = "Hello to All!";
```

Reassignment

Notes on Java Strings

- Simply pass the String to System.out.print or System.out.println

```
String hello = "Hello World!";
String niceDay = "Have a nice day!";
System.out.print(hello);
System.out.print(niceDay);
```

```
Hello World!Have a nice day!
```

Notes on Java Strings

- Simply pass the String to System.out.print or System.out.println

```
String hello = "Hello World!";
String niceDay = "Have a nice day!";
System.out.println(hello);
System.out.println(niceDay);
```

```
Hello World!
Have a nice day!
```

Notes on Java Strings

```
String hello = "Hello ";
String world = "World!";
String helloWorld = hello + world;
System.out.println(helloWorld);
```

```
Hello World!
```

Note the values of the String variables hello and world **do not change**.

Notes on Java Strings

- Literals can be concatenated with the values of String variables.

```
String hello = "Hello ";
String world = "World!";
System.out.println(hello + world
                     + " Have a great day!");
```

```
Hello World! Have a great day!
```

The variables hello, world, and the literal “ Have a great day!” are concatenated together as one String. This one String is then passed to the System.out.println to be printed.

Notes on Java Strings

- You can concatenate primitives into a String.
  - This wasn’t possible in Python without converting int or float types to strings.

```
int days = 31;
System.out.println("There are " + days
                    + " days in January.");
```

```
There are 31 days in January
```

Notes on Java Strings

- You can use the addition augmented assignment operator (+=) to append to a String.

```
String farewell = "See you in ";
farewell += 2 + " days!";
System.out.println(farewell);
```

```
See you in 2 days!
```

Notes on Java Strings

- The toUpperCase method converts the String to a version of itself in all caps.
- Takes no parameters.

```
String hello = "Hello World!";
hello = hello.toUpperCase();
System.out.println(hello);
```

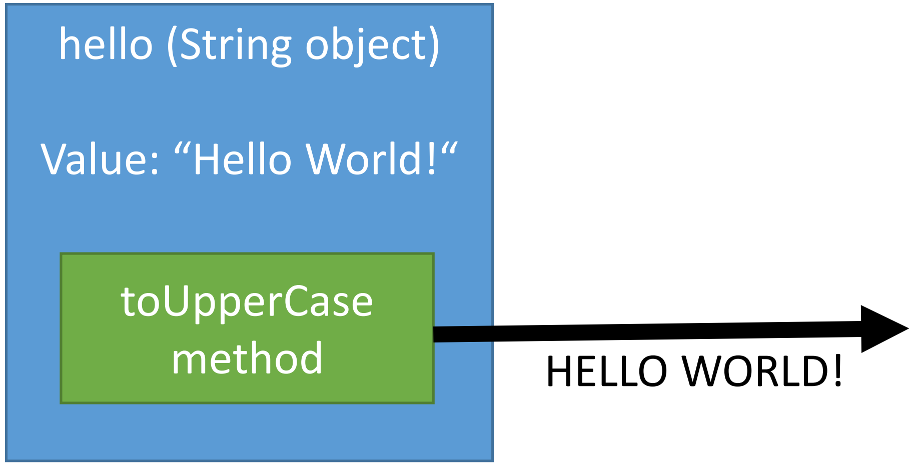
```
HELLO WORLD!
```

Note that we assign hello.toUpperCase() back to the String variable hello. In other words, we are replacing the original value “Hello World!” with “HELLO WORLD!” As you will see in the next slide, the toUpperCase() method doesn’t actually change the String’s value, it just returns the uppercase version of its data.

Notes on Java Strings

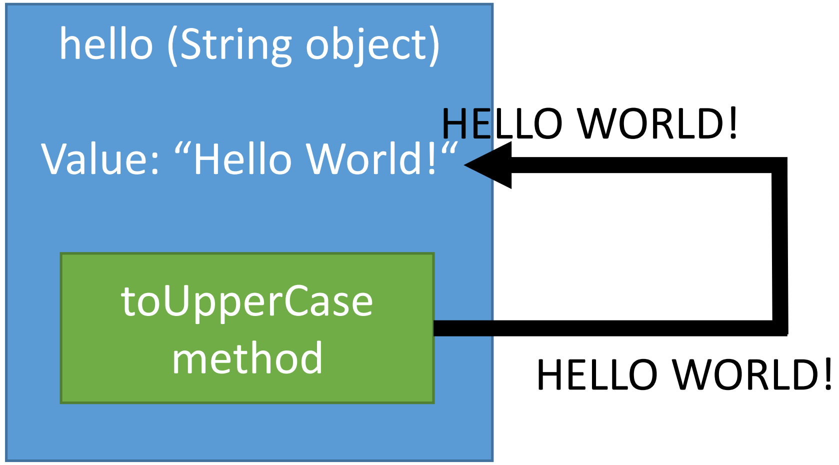
```
String hello = "Hello World!";
```

```
hello.toUpperCase();
```



Doesn’t actually change the String’s value.

```
hello = hello.toUpperCase();
```



Here, we overwrite hello’s initial value by assigning the result of the toUpperCase method back to the String itself.

Notes on Java Strings

```java
String hello = "Hello World!";
System.out.println(hello.toUpperCase());
System.out.println(hello);
```

```
HELLO WORLD!
Hello World!
```

The toUpperCase() method doesn’t actually change the String’s value, it just returns the uppercase version of its data. When we print the value of hello again, it still has its original form (“Hello World!”)

Notes on Java Strings

- The toLowerCase method converts the String to a version of itself in all lowercase.
- Takes no parameters.

```
String hello = "Hello World!";
hello = hello.toLowerCase();
System.out.println(hello);
```

```
hello world!
```

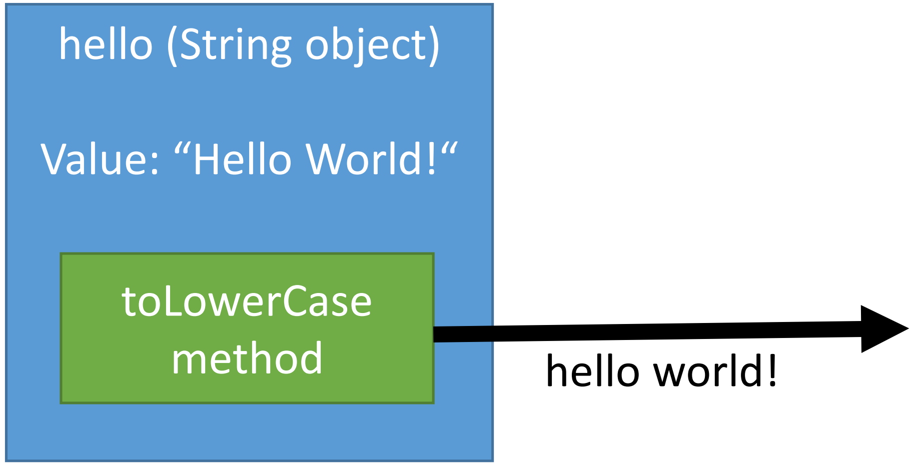
Just like the toUpperCase() method, the toLowerCase() method doesn’t actually change the String’s value, it just returns the lowercase version of its data. In this example, we replaced the original value “Hello World!” with “hello world!”

Notes on Java Strings

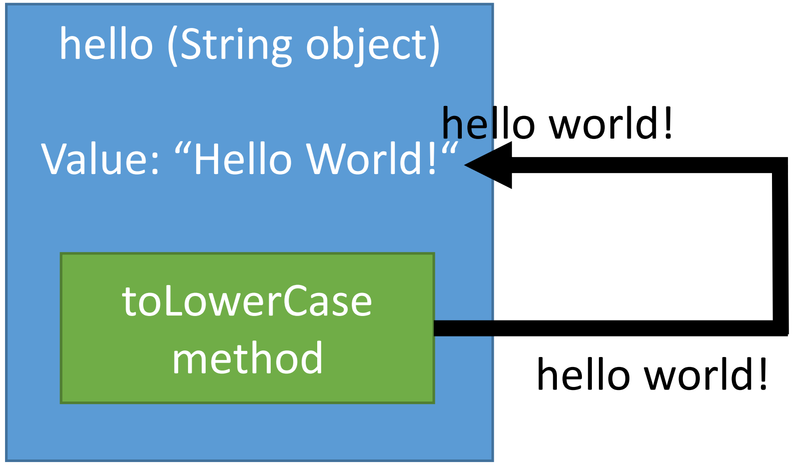
```
String hello = "Hello World!";
```

```
hello.toLowerCase();
```

```
hello = hello.toLowerCase();
```



Doesn’t actually change the String’s value.



Here, we overwrite hello’s initial value by assigning the result of the toLowerCase method back to the String itself.

Escape Sequences in Java

- \n – Line Feed
- \t – Tab
- \” – Double Quote
- \\ – Backslash

- No major differences from Python

Escape Sequences - **\n**

- \n inserts a line feed (or starts a new line)

```
System.out.print("Hello \nWorld");
```

```
Hello
World
```

Escape Sequences - \t

- Inserts one tabspace.

```
System.out.print("Hello \tWorld");
```

```
Hello      World
```

Escape Sequences - \”

- Inserts a double quote character
  - Without this, the compiler will interpret the “ as the start/end of a String literal.

```
System.out.print("\"Hello\" World");
```

```
“Hello” World
```

Escape Sequences - \\

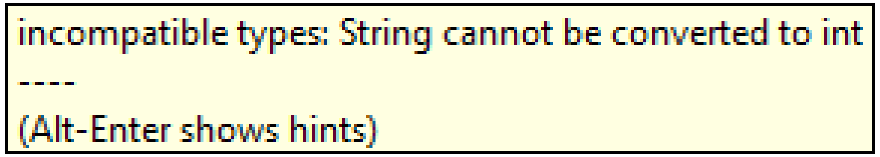
- Inserts a backslash character.
  - The single backslash indicates the start of an escape sequence to the compiler.
  - So, the backslash character itself needs to be escaped.

```
System.out.print("Hello \\ World");
```

```
Hello \ World
```

Parsing Numbers from Strings

```
String ten = "10";
int result = ten + 15;
```

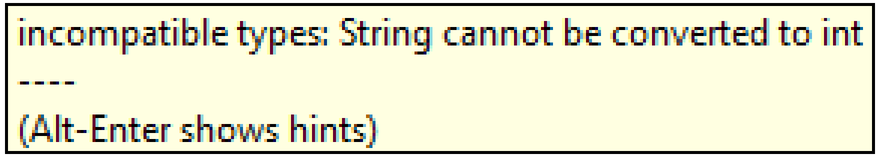


incompatible types: String cannot be converted to int

(Alt-Enter shows hints)

- The above code will not work! You cannot perform arithmetic with Strings, even if the String's characters are numbers.
    - The result of **ten + 15** would be **"1015"** not **25**. Java would treat this as concatenation since one of the operands is a String.
- You also cannot directly assign a String to a numeric primitive like an int, float, etc even if the String's value is a number.

```
int ten = "10";
```



incompatible types: String cannot be converted to int

(Alt-Enter shows hints)

- Numbers must be *parsed* out of a String before you can use them as numeric values.

Parsing Numbers from Strings

- The Integer object is a special type of object known as a “wrapper” object.
  - Essentially, the Integer object has an int inside of it… or, “wraps” around the int.

- A primitive like int doesn’t have the capability of having methods. However, having it wrapped in an object will add that capability.

- The Integer object has a method called parseInt that takes in a String as a parameter, attempts to convert that String’s value as a number, and returns it in int form.

Parsing Numbers from Strings

```java
String ten = "10";
int result = Integer.parseInt(ten) + 15;
System.out.println("The result is " + result);
```

```
The result is 25
```

Parsing Numbers from Strings

- Similar to how the Integer object wraps around an int, the Double object is a wrapper object for the double primitive data type.

- Also like the Integer object, the Double object has a method called parseDouble that takes in a String as a parameter, attempts to convert that String's value to a number, and returns it in double form.

Parsing Numbers from Strings

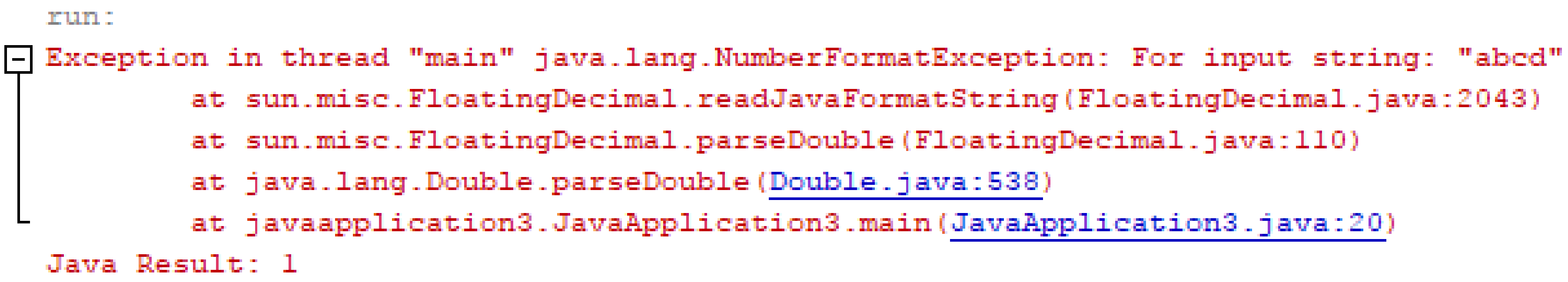
```java
String tenFive = "10.5";
double result = Double.parseDouble(tenFive) + 15;
System.out.println("The result is " + result);
```

```
The result is 25.5
```

Parsing Numbers from Strings

- A **NumberFormatException** is a runtime error that will occur when you try to convert a String that isn't a number into a number.

```
String letters = "abcd";
double myDouble = Double.parseDouble(letters);
System.out.println(myDouble);
```



```
run:
Exception in thread "main" java.lang.NumberFormatException: For input string: "abcd"
        at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
        at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
        at java.lang.Double.parseDouble(Double.java:538)
        at javaapplication3.JavaApplication3.main(JavaApplication3.java:20)
Java Result: 1
```

Python and Java Syntax/Statement Comparisons

Purpose	Python	Java	Result/Output
Assignment	`x = 1`	`x = 1;`	
Print (with line feed)	`print("example")` `print("example")`	`System.out.println("example");` `System.out.println("example");`	example example
Print (with no line feed)	`print("example", endl="")` `print("example", endl="")`	`System.out.print("example");` `System.out.print("example");`	exampleexample

Strings	`test1 = 'abc'`
	`test2 = "def"`
		`test1 = "abc"`	

Concatenation
(String with Strings)
	`test1 + "xyz"`
	`test1 + 'xyz'`
		`test1 + "xyz";`	

Concatenation (String with other types)	`test1 + str(q)`	`test1 + q;`	

Appending
(String with String)
	`test1 += "xyz"`
	`test1 = test1 + "xyz"`
		`test1 += "xyz";`
		`test1 = test1 + "xyz";`	

Appending
(String with other type)
	`test1 += str(q)`
	`test1 = test1 + str(q)`
		`test1 += q;`
		`test1 = test1 + q;`	

Python and Java Syntax/Statement Comparisons

Purpose	Python	Java
Typecast (String to int)	`int(string)`	`Integer.parseInt(string)`
Typecast (String to double)	`float(string)`	`Double.parseDouble(string)`

Keyboard Input	`value = input("Prompt: ")`	`System.out.print("Prompt: ");`
		String value = *<Scanner>*.nextLine();

Single Line Comments	`# Comment`	`// Comment`

Multi-line Comments	`""" Comment """`
	`''' Comment '''`
		`/* Comment */`