# Programming 1

## Week 10 – User-defined Classes (Part 2)

# Object-oriented Programming

In the previous lecture, we have seen how to define a simple class

Usually in a class we will have:

- Data member
- Constructor
- Specific method
- Getter and setter

Beyond those members, we will learn other commonly used methods

- toString()
- equals()

# Review of Instance Fields and Methods

- Each instance of a class has its own copy of instance variables.
  - Example:
    - The `Rectangle` class defines a `length` and a `width` field.
    - Each instance of the `Rectangle` class can have different values stored in its `length` and `width` fields.
- Instance methods require that **an instance of a class** be created in order to be used.
- Instance methods typically interact with instance fields or calculate values based on those fields.

# The `toString` Method

- All objects have a `toString` method that returns the class name and a hash of the memory address of the object.
- We can override the default method with our own to print out more useful information.

```
public String toString() {
        return String.format(" %s %d", name, age);
}
```

# The `toString` Method

- `toString`() method generates a string to represent the object, usually it returns the value of each data member of the object.

- `toString`() method will automatically be called when we want to print an object of that class.

- Once you add the `toString`() method to your class and you print any object of that class, instead of printing the address, it will print a string as you defined in the `toString`() method.

# The `toString` Method

- The `toString` method of a class can be called **explicitly**:

```
Student s1 = new Student (123,"Sanad Adam", 91.50);
System.out.println(s1.toString());
```

- However, the `toString` method does not have to be called explicitly but is called implicitly whenever you pass an object of the class to `println` or `print`.

```
Student s1 = new Student (123,"Sanad Adam", 91.50);
System.out.println(s1);
```

# Demo

```
public String toString() {
    String str = "";
    str += String.format("%-10s: %s\n", "ID", id);
    str += String.format("%-10s: %s\n", "Name", name);
    str += String.format("%-10s: %d\n", "Age", age);
    return str;
}
```

# The `equals` Method

- When the == operator is used with reference variables, the memory address of the objects are compared.
- The contents of the objects are not compared.
- All objects have an `equals` method.
- The default operation of the `equals` method is to compare memory addresses of the objects (just like the == operator).
- We don’t care whether two objects have the same address or not, we care about if the two objects contain the same data members or not.

# The `equals` Method

- The `Student` class has an `equals` method.
- If we try the following:

```
Student s1 = new Student (”Sanad Adam”, 20);
Student s2 = new Student (“Sanad Adam”, 20);
if (s1 == s2) // This is a mistake.
  System.out.println("The objects are the same.");
else
  System.out.println("The objects are not the same.");
```

- only the addresses of the objects are compared.

# The `equals` Method

- Thus, instead of using the == operator to compare two `Students` objects, we should use the `equals` method.

```
public boolean equals(Student object2)
{
   boolean status;

   if(name.equals(Object2.name && age == Object2.age)
      status = true;
   else
      status = false;
   return status;
}
```

- Now, objects can be compared by their contents rather than by their memory addresses.
- Here the name is a String variable, so to compare the name we also need to use the equals() method.

# The `equals` Method

- `equals`() will take one object of the same class as a parameter, then check if all the data members of that object are the same as the current object.

- If every object are equal, then we say this two objects are exactly the same, and we return true. However, if at least one data member is not the same, we say the two object are not equal.

- Remember, == should only be used to compare two primitive data type values, for example int, char, boolean, double etc.

# Demo

# Shallow Copy VS Deep Copy

- There are two ways to copy an object.
  - *You cannot use the assignment operator to copy reference types*
  - Reference only copy
    - This is simply copying the address of an object into another reference variable.
  - Deep copy (correct)
    - This involves creating a new instance of the class and copying the values from one object into the new object.

# Shallow Copy VS Deep Copy

- If we have two primitive variables, and we want to give the second variable's value to the first variable, we can just use the "=".

```
int num1 = 1;
int num2 = 2;
num1 = num2
```

- After these three lines of code, we will have
  - num1 equals 2
  - num2 equals 2

    num1 and num2 are two different variables.

# Copy Constructors

- A copy constructor accepts an existing object of the same class and clones it
- The copy constructor is a constructor that takes another object of the same class as parameter, and then initialize each data member of the new object as the other object

```
public Student(Student object2)
{
      name = object2.name;
      age = object2.age;
}

// Create a Student object
Student s1 = new Student(“Adma", 20);

//Create S2, a copy of s1
Student s2 = new Student(s1);
```

# Static Class Members

- **Static fields** and **static methods** do not belong to a single instance of a class.
- A static data member or method belongs to the class.
- A non-static data member or method belongs to the object
- To invoke a static method or use a static field, the class name, rather than the instance name, is used.
- Example:

```
double val = Math.sqrt(25.0);
```

# Static Fields

- Class fields are declared using the `static` keyword between the access specifier and the field type.

```
private static int instanceCount = 0;
```

- The field is initialized to 0 only once, regardless of the number of times the class is instantiated.
  - Primitive static fields are initialized to 0 if no initialization is performed.

# static vs non-static

- Every time when you create a new object, Java will provide a new piece of memory to store its non-static data members.
- If we have 1000 students, there will be 1000 names, 1000 age and 1000 emails
- Since a non-static data member belongs to each object, we should initialize it inside constructors.
- Java will only create one piece of memory to store a static data member. Then when you create a new object, Java will not provide a new piece of memory for the static data member again.
- For Example, if we have 1000 students, there will be 1000 names, 1000 age and 1000 emails, and only 1 college.

- Since static a data member belongs to the class, we should initialize it once we create it.

# static vs non-static

- Since static data member belongs to the class, then all objects of that class share that value.

```
public class Museum {
     private static int visitorNum = 0;        //
static

     public void increaseVisitNum() {
                 visitorNum++;
        }
}
```

- If any object changes a static data member, it will also affect other objects.

```
public static void main(){
   Museum m1 = new Museum();  // visitorNum = 0
   m1.increaseVisitNum();              // visitorNum =
1
   Museum m2 = new Museum();  // visitorNum = 1
   m2.increaseVisitNum();              // visitorNum = 2
}
```

# Static Methods

- Methods can also be declared static by placing the `static` keyword between the access modifier and the return type of the method.

```
public static double milesToKilometers(double miles)
{…}
```

- When a class contains a static method, it is not necessary to create an instance of the class in order to use the method.

# static vs non-static

- A method can also be static, but a static method can only visit static data member and cannot visit non-static data members.
- In this case, most of the methods in a class should be non-static.
- If a method is on a class level, and only need access to static data members but no non-static data members, we can make it static.

```
public class Student {
        private static String college = “Vanier”; // static
        private String name; // non-static
        private int age; // non-static
        private String email; // non-static

        public static String getCollege() {
             return college; // only visit static data member
        }

        public String getName() {
             return name; // visit non-static data member
        }

}
```

# Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes, such as the `Math` class in the Java Standard Library.
- Static methods may not communicate with instance fields, only static fields.

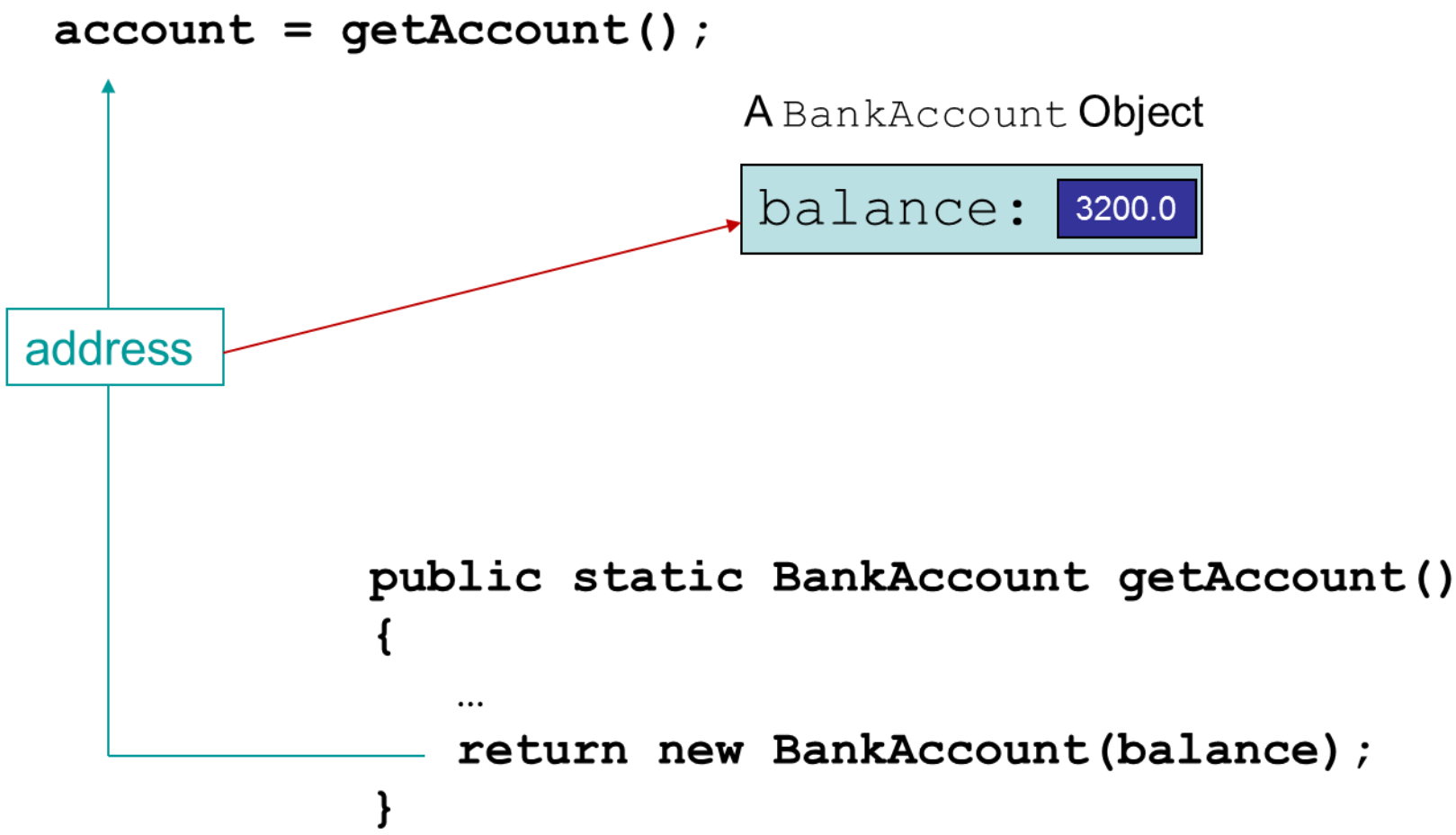
# Demo

# Returning Objects From Methods (1 of 2)

- Methods are not limited to returning the primitive data types.
- Methods can return references to objects as well.
- Just as with passing arguments, a copy of the object is **not** returned, only its address.
- Method return type:

```
public static BankAccount getAccount()
{
    …
     return new BankAccount(balance);
}
```

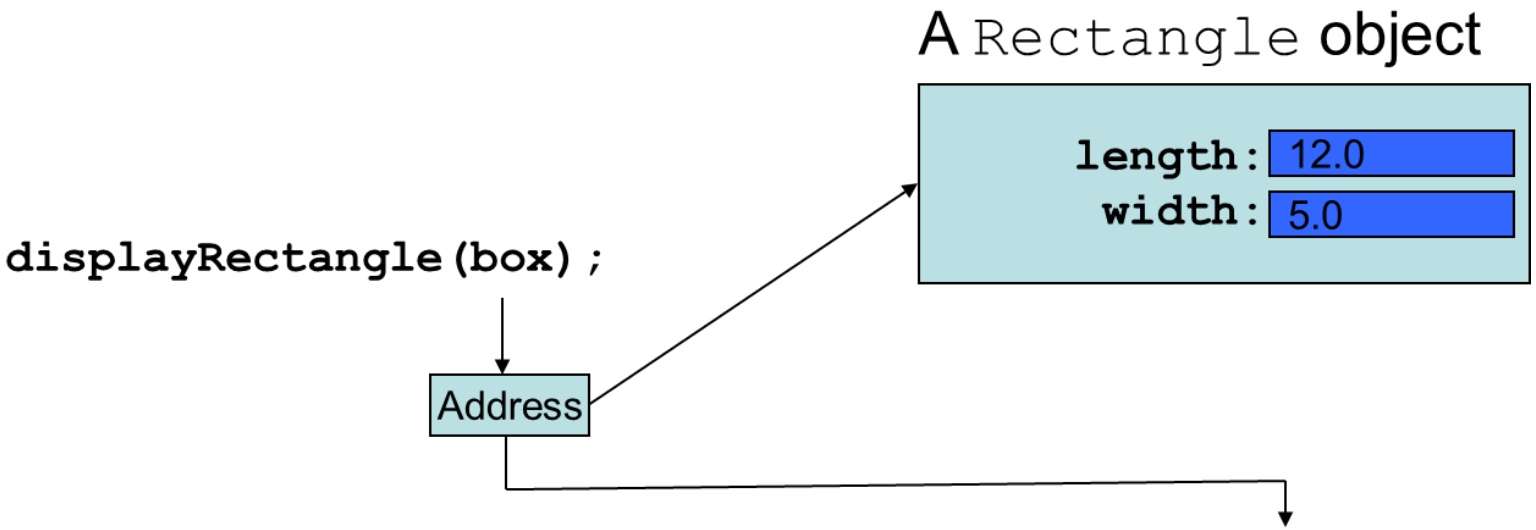
# Returning Objects From Methods (2 of 2)



# Passing Objects as Arguments (1 of 2)

- Objects can be passed to methods as arguments.
- Java passes all arguments **by value**.
- When an object is passed as an argument, the value of the reference variable is passed.
- The value of the reference variable is an address or reference to the object in memory.
- A **copy** of the object is **not passed**, just a pointer to the object.
- When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable.

# Passing Objects as Arguments (2 of 2)




```
public static void displayRectangle(Rectangle r)
{
  // Display the length and width.
  System.out.println("Length: " + r.getLength() +
                     " Width: " + r.getWidth());
}
```

# Returning References to Private Fields

- Avoid returning references to private data elements.
- Returning references to private variables will allow any object that receives the reference to modify the variable.

# Null References

- A **null reference** is a reference variable that points to nothing.
- If a reference is null, then no operations can be performed on it.
- References can be tested to see if they point to null prior to being used.

```
if(name != null)
{
  System.out.println("Name is: " + name.toUpperCase());
}
```

# The `this` Reference

- The `this` reference is simply a name that an object can use to refer to itself.
- The `this` reference can be used to overcome shadowing and allow a parameter to have the same name as an instance field.