Day 02: Using Classes and Objects

Slide Credits: *Internet / Chetan Arora*

Where did all these instances come from?

- We know how to send messages to an instance of a class by calling methods
- So far, we've called methods on **samBot**, an instance of `Robot`
- Where did those classes come from?
- Next: how to use a class as a blueprint to actually build instances!

Constructors (1/3)

- `Robot`s can turnRight, moveForward, etc.
- Can call any of these methods on any instance of `Robot`
- But how did these instances get created in the first place?
- Define a special kind of method in the `Robot` class: a **constructor**
- **Note: every object must have a constructor**

```
public class Robot {

  public void turnRight() {
    // code that turns robot right
  }

  public void moveForward(int numSteps) {
    // code that moves Robot forward
  }

  /* other code */
}
```

Constructors (2/3)

- A **constructor** is a special kind of method that is called whenever an object is to be “born”, i.e., created – see shortly how it is called
- Constructor’s name is always the same as name of class
- If the class is called “Robot”, its constructor needs to be called “Robot”. If the class is called “Dog”, its constructor had better be called “Dog”

```
public class Robot {

public Robot () {
    // this is the constructor!
}

public void turnRight() {
   // code that turns robot right
}

public void moveForward(int numSteps) {
   // code that moves Robot forward
}

/* other code */
}
```

Constructors (3/3)

- Constructors are special methods: used only once, to create the instance
- And we never specify a return value in its declaration
- Constructor for `Robot` does not take in any parameters (notice empty parentheses)
- Constructors can, and often do, take in parameters– later…

```
public class Robot {

public Robot () {
    // this is the constructor!
}

public void turnRight() {
   // code that turns robot right
}

public void moveForward(int numSteps) {
   // code that moves Robot forward
}

/* other code */
}
```

Instantiating Objects (1/3)

- Now that the `Robot` class has a constructor, we can create instances of `Robot` !
- Here's how we create a `Robot` in Java:

```
new Robot();
```

- This means "use the `Robot` class as a blueprint to create a new `Robot` instance"
- `Robot()` is a call to `Robot`'s constructor, so any code in the constructor will be executed as soon as you create a `Robot`

Instantiating Objects (2/3)

- We refer to “creating” an object as **instantiating** it
- When we say:

```
new Robot();
```

- … We’re **creating an instance** of the `Robot` class, a.k.a. **instantiating** a new `Robot`
- Where exactly does this code get executed?

Invoking Methods

- We've seen that once an object has been instantiated, we can use the *dot operator* to invoke its methods

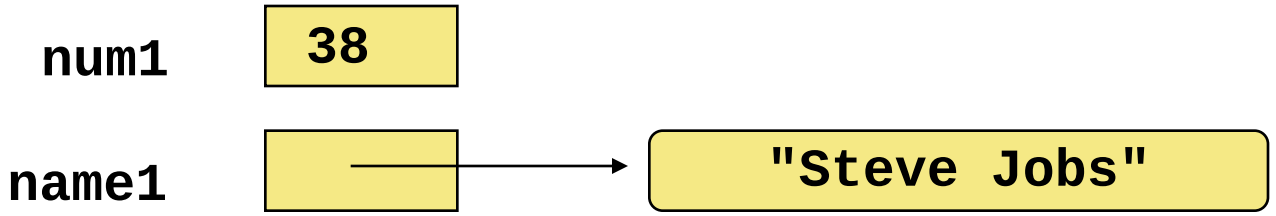
```
count = title.length()
```

- A method may *return a value*, which can be used in an assignment or expression

- A method invocation can be thought of as asking an object to perform a service

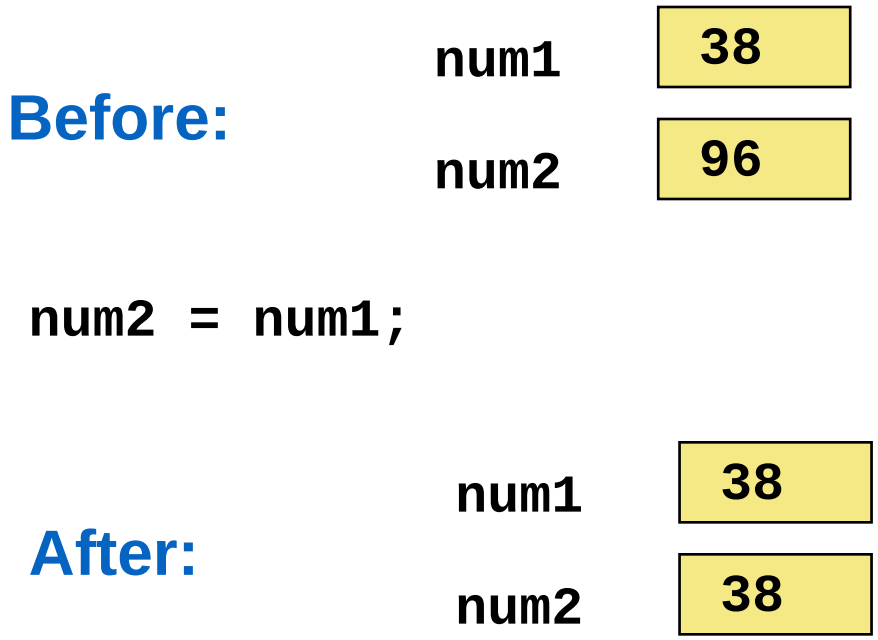
References

- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically



Assignment Revisited

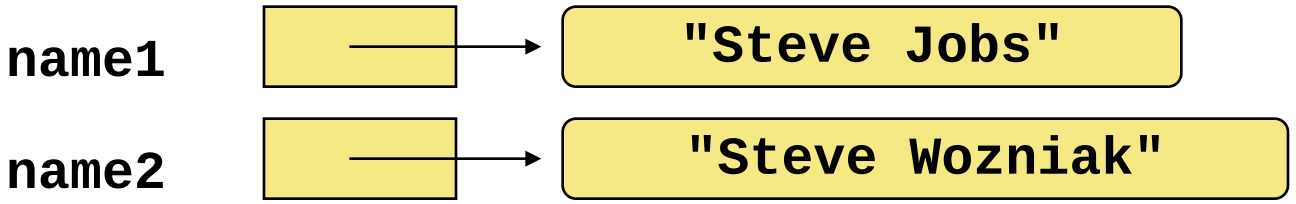
- The act of assignment takes a copy of a value and stores it in a variable
- For primitive types:



Reference Assignment

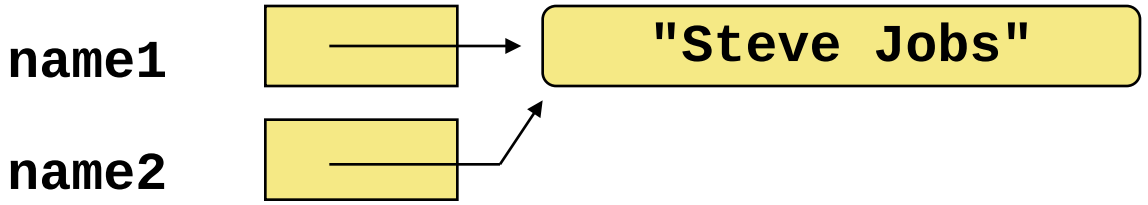
- For object references, assignment copies the address:

Before:




```
name2 = name1;
```

After:



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- That creates an interesting situation: one object can be accessed using multiple reference variables
- Aliases can be useful, but should be managed carefully
- Changing an object through one reference changes it for all of its aliases, because there is really only one object

Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer is responsible for performing garbage collection

Class Libraries

- A *class library* is a collection of classes that we can use when developing programs
- The *Java standard class library* is part of any Java development environment
- Its classes are not part of the Java language per se, but we rely on them heavily
- Various classes we've already used (`System`, `String`) are part of the Java standard class library
- Other class libraries can be obtained through third party vendors, or you can create them yourself

Packages

- The classes of the Java standard class library are organized into *packages*
- Some of the packages in the standard class library are:

Package	Purpose
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities
javax.xml.parsers	XML document processing

The import Declaration

- When you want to use a class from a package, you could use its *fully qualified name*

```
java.util.Scanner;

java.util.Random;
```

Or you can *import* the class, and then use just the class name

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

- To import all classes in a particular package, you can use the * wildcard character

```
import java.util.*;
```

The import Declaration

- All classes of the `java.lang` package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the `System` or `String` classes explicitly in earlier programs
- The `Scanner` class, on the other hand, is part of the `java.util` package, and therefore must be imported

Class based Access Privileges: Access Modifiers

- Determine whether other classes can use a particular field or invoke a particular method
- The first data column indicates whether the class itself has access to the member defined by the access level.
- The second column indicates whether classes in the same package as the class have access to the member.
- The third column indicates whether subclasses of the class declared outside this package have access to the member.
- The fourth column indicates whether all classes have access to the member.

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
`no modifier`	Y	Y	N	N
`private`	Y	N	N	N

The Random Class

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudorandom numbers
- A `Random` object performs complicated calculations based on a *seed value* to produce a stream of seemingly random values

```
      Random generator = new Random();
int num1;
float num2;

num1 = generator.nextInt();
System.out.println ("A random integer: " + num1);

num1 = generator.nextInt(10);
System.out.println ("From 0 to 9: " + num1);

num1 = generator.nextInt(15) + 20;
System.out.println ("From 20 to 34: " + num1);

num1 = generator.nextInt(20) - 10;
System.out.println ("From -10 to 9: " + num1);

num2 = generator.nextFloat();
System.out.println ("A random float [between 0-1]: " + num2);

num2 = generator.nextFloat() * 6;  // 0.0 to 5.999999
num1 = (int) num2 + 1;
System.out.println ("From 1 to 6: " + num1);
```

Interactive Programs

- Programs generally need input on which to operate
- The `Scanner` class provides convenient methods for reading input values of various types
- A `Scanner` object can be set up to read input from various sources, including the user typing values on the keyboard
- Keyboard input is represented by the `System.in` object

Reading Input

- The following line creates a Scanner object that reads from the keyboard:

```
Scanner scan = new Scanner (System.in);
```

- The `new` operator creates the `Scanner` object
- Once created, the `Scanner` object can be used to invoke various input methods, such as:

```
answer = scan.nextLine();
```

Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used
- The `nextLine` method reads all of the input until the end of the line is found

Echo.java

```
import java.util.Scanner;
public class Echo {
      //  Reads a character string from the user and prints it.
      public static void main (String[] args) {
       String message;
       Scanner scan = new Scanner (System.in);

       System.out.println ("Enter a line of text:");
       message = scan.nextLine();

       System.out.println ("You entered: \"" + message + "\"");
   }
}
```

Input Tokens

- Unless specified otherwise, *white space* is used to separate the elements (called *tokens*) of the input
- White space includes space characters, tabs, new line characters
- The `next` method of the `Scanner` class reads the next input token and returns it as a string
- Methods such as `nextInt` and `nextDouble` read data of particular types

```
public static void main (String[] args)   {
    int miles;
    double gallons, mpg;

    Scanner scan = new Scanner (System.in);

    System.out.print ("Enter the number of miles: ");
    miles = scan.nextInt();

    System.out.print ("Enter the gallons of fuel used: ");
    gallons = scan.nextDouble();

    mpg = miles / gallons;

    System.out.println ("Miles Per Gallon: " + mpg);
 }
```

Static Class Members

- A static method can be invoked through its class name
- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Variables can be static as well
- Determining if a method or variable should be static is an important design decision

The static Modifier

- We declare static methods and variables using the `static` modifier
- It associates the method or variable with the class rather than with an object of that class
- Static methods are sometimes called *class methods* and static variables are sometimes called *class variables*
- Let's carefully consider the implications of each

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, only one copy of the variable exists

```
private static float price;
```

- Memory space for a static variable is created when the class is first referenced
- All objects instantiated from the class share its static variables
- Changing the value of a static variable in one object changes it for all others

Static Methods

```
class Helper
{
   public static int cube (int num)
   {
      return num * num * num;
   }
}
```

Because it is declared as static, the method can be invoked as

```
value = Helper.cube(5);
```

Static Class Members

- The order of the modifiers can be interchanged, but by convention visibility modifiers come first
- Recall that the `main` method is static – it is invoked by the Java interpreter without creating an object
- Static methods cannot reference instance variables because instance variables don't exist until an object exists
- However, a static method can reference static variables or local variables

Static Class Members

- Static methods and static variables often work together
- The following example keeps track of how many objects have been created using a static variable, and makes that information available using a static method

```
class MyClass {
   private static int count = 0;

   public MyClass () {
      count++;
   }
   public static int getCount () {
      return count;
   }
}
```

```
      MyClass obj;

      for (int scan=1; scan <= 10; scan++)
         obj = new MyClass();

      System.out.println ("Objects created: " +
       MyClass.getCount());
```

Student Id problem

- Let’s suppose we have a Student class
- How do we assign unique student ids to each student object that we create?
- What if we also want to get the latest Student created? Like:

```
public static String getLatestStudent()
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

Java Final Variable

- If you make any variable as final, you cannot change the value of final variable(It will be constant).

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
```

Java Final Method

- If you make any method as final, you cannot override it.

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

Java Final Class

If you make any class as final, you cannot extend it.

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

The this Reference

- The `this` reference allows an object to refer to itself
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- **In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2`**

The this reference

- The `this` reference can be used to distinguish the instance variables of a class from corresponding method parameters with the same names
- The constructor of the `Account` class (from Chapter 4) could have been written as follows:

```
public Account (Sring name, long acctNumber,
                double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

References

- Recall that an object reference holds the memory address of an object
- Rather than dealing with arbitrary addresses, we often depict a reference graphically as a “pointer” to an object

```
ChessPiece bishop1 = new ChessPiece();
```

References

- Things you can do with a reference:
    - Declare it : String st;
    - Assign a new value to it
        - st = new String(“java”);
        - st = st2;
        - st = null;
    - Interact with the object using “dot” operator : st.length()
    - Check for equivalence
        - (st == st2)
        - (st == null)

The null Reference

- An object reference variable that does not currently point to an object is called a *null reference*
- The reserved word `null` can be used to explicitly set a null reference:

```
name = null;
```

or to check to see if a reference is currently null:

```
if (name == null)
   System.out.println ("Invalid");
```

The null Reference

- An object reference variable declared at the class level (an instance variable) is automatically initialized to null
- The programmer must carefully ensure that an object reference variable refers to a valid object before it is used
- Attempting to follow a null reference causes a `NullPointerException` to be thrown
- Usually a compiler will check to see if a local variable is being used without being initialized

Objects as Parameters

- Another important issue related to method design involves parameter passing
- Parameters in a Java method are *passed by value*
- A copy of the argument (the value passed) is stored into the formal parameter (in the method header)
- Therefore passing parameters is similar to an assignment statement
- When an object is passed to a method, the argument and the formal parameter become aliases of each other

Passing Objects to Methods

- What a method does with a parameter may or may not have a permanent effect (outside the method)
- Note the difference between changing the internal state of an object versus changing which object a reference points to

ParameterPassing

```
ParameterTester tester = new ParameterTester();

    int a1 = 111;
    Num a2 = new Num (222);
    Num a3 = new Num (333);

    System.out.println ("Before calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");

    tester.changeValues (a1, a2, a3);

    System.out.println ("After calling changeValues:");
    System.out.println ("a1\ta2\ta3");
    System.out.println (a1 + "\t" + a2 + "\t" + a3 + "\n");
```

ParameterTester

```
class ParameterTester
{
//  Modifies the parameters, printing their values before and
   //  after making the changes.
   //-----------------------------------------------------------------
   public void changeValues (int f1, Num f2, Num f3)
   {
      System.out.println ("Before changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");

      f1 = 999;
      f2.setValue(888);
      f3 = new Num (777);
      System.out.println ("After changing the values:");
      System.out.println ("f1\tf2\tf3");
      System.out.println (f1 + "\t" + f2 + "\t" + f3 + "\n");
   }
}
```

Num

```
class Num {
   private int value;

public Num (int update) {
      value = update;
   }

public void setValue (int update)
   {
      value = update;
   }

public String toString () {
      return value + "";
   }
}
```

Method Overloading

- *Method overloading* is the process of giving a single method name multiple definitions
- If a method is overloaded, the method name is not sufficient to determine which method is being called
- The *signature* of each overloaded method must be unique
- The signature includes the number, type, and order of the parameters

Method Overloading

- The compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
   return x + .375;
}

float tryMe(int x, float y)
{
   return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```

Method Overloading

- The `println` method is overloaded:

```
println (String s)
println (int i)
println (double d)
```

and so on...

- The following lines invoke different versions of the `println` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Overloading Methods

- The return type of the method is not part of the signature
- That is, overloaded methods cannot differ only by their return type

Overloading Methods

- Constructors can be overloaded
- An overloaded constructor provides multiple ways to set up a new object

Thanks!