

# ESS 201: Programming II

## Java

Term 1, 2020-21

Classes - 2

T K Srikanth  
IIIT-B

# Class definition

```
class IceCreamBar {  
    IceCreamBar(String iname, String iflavour, float itemp, Constructor  
                float iprice) { }  
  
    String getName()      { }  
  
    String getFlavour()   { }  
  
    float getTemp()       { }  
  
    float getPrice()      { }  
  
    String name, flavour;  
  
    float temp, price;  
}
```

**Methods or “behaviour”**

**Data members or “State”**

# Constructors

Every class should have at least one constructor. Constructors are responsible for creating an instance of that class, and should ensure that the created object is in a consistent and correct state.

Constructor that takes no arguments is called the *default constructor*

Compiler defines a default constructor if no constructors have been defined for the class. If any constructor is defined for the class, then the constructor does not define the default constructor

All instance variables are initialized to *default* values, if not explicitly initialized in a constructor. The default version sets all data members to their *default initial value*.

Note: no notion of “*destructors*”. Facility for cleanup

# Class definition

```
class IceCreamBar {  
  
    IceCreamBar(String iname, String iflavour, float itemp,  
float iprice) { ... }  
  
    String getName() { return name; }  
  
    String getFlavour() { return flavour; }  
  
    float getTemp() { return temp; }  
  
    float getPrice() { return price; }  
  
    ...  
  
}
```

# Invoking class methods

```
IceCreamBar ic1 = new IceCreamBar("CH", "Chocolate", 8.0, 90.0);
```

...

```
System.out.println("Flavour " + ic1.getFlavour() +  
                    " from " + ic1.getName() +  
                    " costs " + ic1.getPrice() +  
                    " and is stored at " +  
                    ic1.getTemp() + " degrees.");
```

*Should print:*

Flavour Chocolate from CH costs 90.0 and is stored at 8.0  
degrees.

**Note:** any invocation of a method of some class always is from within some method of the same or other class. No free standing executions/statements. Even **main** is a method of some class

# Class definition

```
class IceCreamBar {  
    IceCreamBar(String iname, String iflavour, float itemp,  
float iprice) {  
        name = iname; flavour = iflavour;  
        temp = itemp;  
        price = iprice;  
    }  
    String getName() { return name; }  
    String getFlavour() { return flavour; }  
    float getTemp() { return temp; }  
    float getPrice() { return price; }  
    ...  
}
```

# Method signatures and overloading

The name and parameter list (arguments and their types) defines the *signature* of a method

## Method overloading

- we can have multiple methods in a class with the same name, but with different arguments. E.g. constructors with different sets of arguments

When an overloaded method is invoked, the compiler maps this to that method whose signature most closely matches the invocation

- Number and order of arguments must match, and the types of arguments should be “compatible” (to be discussed later)

Can we have the following two methods in the same class?

```
class Book {  
    public static int    averageCost(Book[] books);  
    public static double arverageCost(Book[] books);  
}
```

Is the return type part of the signature?



# Overloading

What happens if we have the following methods:

```
void f1(int x) { }
```

```
void f1(long x) { }
```

and we invoke:

```
f1(7);
```

```
f1(7L);
```

Would this work? If so, which method would be called?

What if there is only one definition of f1

# Primitive wrapper classes

primitive variables are not objects - i.e. not instances of some class

Java provides “wrapper” classes corresponding to each primitive type: Byte, Short, Integer,, Float, Double, ... , BigInteger

Compiler supports automatic conversion between primitives and their corresponding wrapper classes: auto boxing and unboxing (like an implicit cast).

- Passed as a parameter to a method that expects an object of the corresponding wrapper class.
- Assigned to a variable of the corresponding wrapper class.

# Auto boxing and unboxing

```
int i=3;  
Integer j = i;  
int k = j;
```

-----

```
void m1(Integer i, Character c) { ... }
```

Can be invoked as:

```
int i=3;  
char c 'a';  
m1(i, c);
```

# Scope and lifetime (of primitives, references, objects)

The scope of primitives and references defined by the block where they are declared. And so is their lifetime (memory no longer available once out of scope)

```
{  
    int x = 7;  
    Box b1 = new Box();  
    ...  
}
```

Objects live on independent of the block where they were constructed. Accessible as long as some reference to them is still alive.

So, how does garbage collection work?

# static methods and data members

A method specified as **static** is considered as a method defined on the class, and not tied to any instance.

Can be invoked even without creating instances of that class

static methods cannot access data (or methods) that are non-static - even of the same class

Similarly, static data members are shared by all instances of that class - there is only one piece of storage associated with that data member.

Static methods/members are accessed with

`Classname.member`

# static methods

```
class Account {  
    ...  
    static Account max(Account[] acs) {}  
    private String name;  
    private float balance;  
}
```

## Usage:

```
Account[] accounts = new  
Account[10];  
  
// ... Initialize array  
  
Account largest =  
  
    Account.max(accounts);
```

# static data fields

```
class Account {  
    ...  
    Account() {  
        accountNumber = nextId++;  
    }  
    static Account max(Account[] acs) {}  
    ...  
    private static int nextId = 1;  
    private String name;  
    private float balance;  
    private int accountNumber;  
}
```

So how does garbage collection work?



# final variables

A variable labelled as **final** implies value cannot be changed once it is initialized

**final** variables must be initialized

- when declared
- or, in every constructor (also called “blank final”)
- Compiler error if attempt to re-assign a value

Note: if a reference to an object is final, the reference itself cannot change, however, the object it refers to can change.

For example, in the IceCreamBar example, the name and flavour should be defined final, if the intent is that these cannot change once the object is initialized.

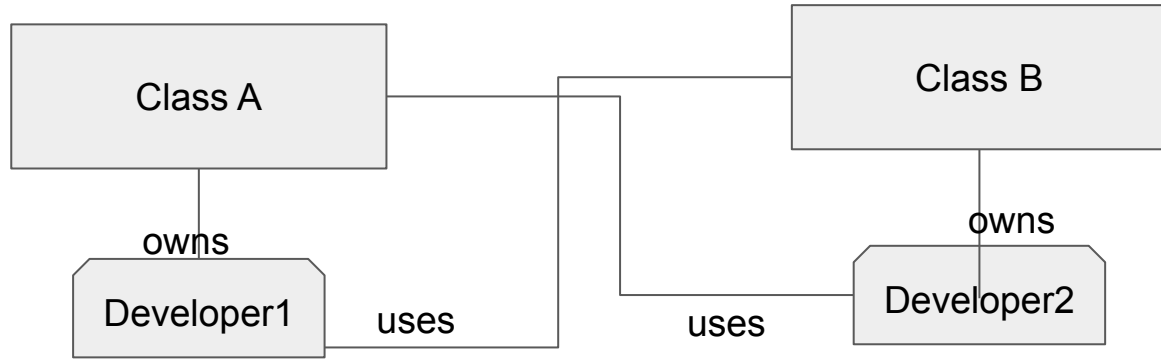
# Hiding the implementation

*“separating the things that change from the things that stay the same.”* - Bruce Eckel, Thinking in Java

The public methods that are used to create/access/modify objects should largely remain the same, although the implementation (including any data members and other classes/methods internal to this implementation) will likely change over time.

Access specifiers are one way to enable this separation.

# Useful design perspective




Note: Developer1 and Developer2 can be the same person.

Owner of a class implements the methods/data members of the class, and should ensure that *any object of that class is always in a correct and consistent state* - starting with the constructors.

The user of a class can access that class only through the methods (and data members) that are *accessible* to this user. That is, the methods of class B can manipulate objects of class A only through the methods (and data members) accessible to this class.

# Class definition

```
class IceCreamBar {  
    IceCreamBar(String iname, String iflavour, float itemp, float iprice)  
    {  
        name = iname; flavour = iflavour;  
        temp = itemp;  
        price = iprice;  
    }  
    String getName()      { return name; }  
    String getFlavour()   { return flavour; }  
    float getTemp()       { return temp; }  
    float getPrice()      { return price; }  
    void setTemp(float t) { temp = t; }  
    void setPrice(float p) { price = p; }  
  
    private String name, flavour;  
    private float temp, price;  
}
```



Cannot be accessed or used in  
a method of a different class

# Invoking class methods - restricted access

What is the outcome of the following statement for the current example?

```
System.out.println("Flavour " + ic1.flavour +  
                    " from " + ic1.name +  
                    "costs " + ic1.price +  
                    " and is stored at " +  
                    ic1.temp + " degrees.");
```

Assume this is a statement in a method

- of a different class (say, IceCreamStore)
- of the same class IceCreamBar