# JAVA Programming

Creating and Destroying objects

# Overview

- Using Constructors
- Initializing Data
- Objects and Memory
- Resource Management

# Using Constructors

- Look at the following example

```
Client client1 = new Client();
client1.setName(“Jean”);
client1.setClientId(100);
```

- The default constructor (no parameters) is called

# Using the Default Constructor

- Features of a default constructor
  - Public accessibility
  - Same name as the class
  - No return type—not even **void**
  - Expects no arguments
  - Initializes all fields to **zero**, **false** or **null**

- Constructors can be overloaded
  - If you create an overloaded constructor, the default constructor is no longer created

# Custom Constructor

- Create a custom constructor

```
Client myclient = new Client("Jean" ,100);
```

```
public Client(String newName, int newId) {
   name = newName;
   id = newId;
}

public void setName(String name) {
    this.name = name;
}
public void setId(int id) {
    this.id = id;
}
```

```
setName(newName);
setId(newId);
```

**Don't Repeat Yourself!**

# Custom Constructor

- After creating a new constructor

```
Client myClient = new Client();
```

won't compile

- Initially each class has a default no-argument constructor
- Recreate default constructor after adding a constructor with parameters

# Custom Constructor

```
public class Client {
 private String name;
 private int id;

 public Client() {
   //add it
   //explicit
 }

 public Client(String newName, int newId) {
   name = newName;
   id = newId;
 }
```

# Constructors

- The constructors are overloaded
- Java uses the signature to decide which constructor to use

# Constructor Chaining

- Constructor chaining
- Default constructor invokes constructor with parameters

```
Client client1 = new Client();
```

```
public Client() {

 this("undefined user", -1);
}

public Client(String newName, int newId) {
  setName(newName);
  setId(newId);
}
```

**Not DRY**

# Using Constructors

- Creating Objects
- Using the Default Constructor
- Overriding the Default Constructor
- Overloading Constructors

# Creating Objects

- Step 1: Allocating memory
  - Use **new** keyword to allocate memory from the heap
- Step 2: Initializing the object by using a constructor
  - Use the name of the class followed by parentheses

```
Date when = new Date( );
```

# Using Private Constructors

- A private constructor prevents objects from being created
  - Instance methods cannot be called
  - Static methods can be called
  - A useful way of implementing procedural functions

```
public class Math{
    public static double Cos(double x) { ... }
    public static double Sin(double x) { ... }
    private Math( ) { }
}
```

# Object Lifetime

- Creating objects
  - You allocate memory by using **new**
  - You initialize an object in that memory by using a constructor
- Using objects
  - You call methods
- Destroying objects
  - The object is converted back into memory
  - The memory is de-allocated

# Objects and Scope

- The lifetime of a local value is tied to the scope in which it is declared
  - Short lifetime (typically)
  - Deterministic creation and destruction
- The lifetime of a dynamic object is not tied to its scope
  - A longer lifetime
  - A non-deterministic destruction

# Garbage Collection

- You cannot explicitly destroy objects
  - Java does not have an opposite of **new** (such as **delete**)
  - This is because an explicit delete function is a prime source of errors in other languages
- Garbage collection destroys objects for you
  - It finds unreachable objects and destroys them for you
  - It finalizes them back to raw unused heap memory
  - It typically does this when memory becomes low

# Object Cleanup

- The final actions of different objects will be different
  - They cannot be determined by garbage collection.
  - Objects in Java have a **finalize** method.
  - If present, garbage collection will call the finalizer before reclaiming the raw memory.
  - In Java implement a finalizer to write cleanup code.
  - Use for instance when holding a handle to native resources like sockets, file handles, window handles, etc.

# Finalizers

```
protected void finalize() throws Throwable {
        // TODO Auto-generated method stub
        super.finalize();
}
```

# Warnings About Destruction Timing

- The order and timing of destruction is undefined
  - Not necessarily the reverse of construction
- Finalizers are guaranteed to be called
  - Cannot rely on timing
- Avoid finalizers if possible
  - Performance costs
  - Complexity
  - Delay of memory resource release

# Lab: Constructors and Resources