

---

# Programming overview

## Part II

---

# Parameter passing in C++

## Call by value

```
int x = 0;  
giveMeATen (x);  
printf ("%d", x);  
[...]  
void giveMeATen (int y) {  
    y = 10;  
}
```

Output: 0

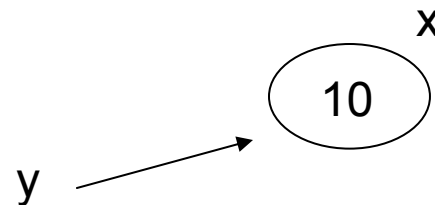
X: 0

Y:10

## Call by reference

```
int x = 0;  
giveMeATen(x);  
printf ("%d", x);  
[...]  
void giveMeATen (int &y) {  
    y = 10;  
}
```

Output: 10



---

## Why do we need to call a function by reference?

- Sometimes we need to get more than one value from a function.
  - Sometimes we need to pass huge data structures to a function. Calling by value in this case is too slow because the system needs to copy contents of the data structure to the parameters of the function.
-

# Parameter passing in Java - by reference or by value?

```
int x = 0;  
giveMeATen (x);  
System.out.println (x);  
[...]  
void giveMeATen (int y) {  
    y = 10;  
}
```

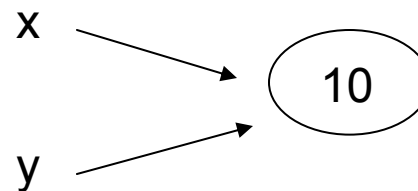
Output: 0

X: 0

Y: 10

```
aClass x = new aClass(0);  
giveMeATen (x);  
System.out.println (x.m);  
[...]  
void giveMeATen (aClass y) {  
    y.m = 10;  
}
```

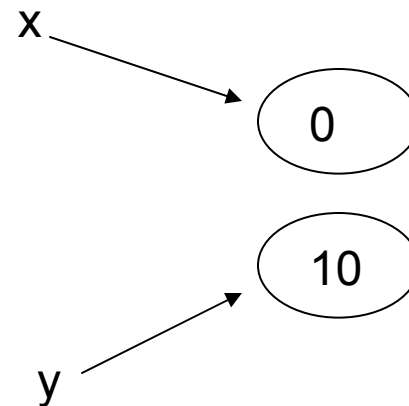
Output: 10



**Myth: "Objects are passed by reference, primitives are passed by value"**

```
aClass x = new aClass(0);  
aClass y = new aClass(10);  
swap (x, y);  
System.out.println(x.m+" " +y.m);  
[...]  
void swap (aClass p, aClass q){  
    aClass temp=p;  
    p = q; q = temp;  
}
```

Output: 0 10



---

# Parameter passing in Java-The truth

- **Truth #1: Everything in Java is passed by value.**
  - **Truth #2: The values of variables are always primitives or references to objects, never objects.**
-

# Primitive variables vs. Reference Variables

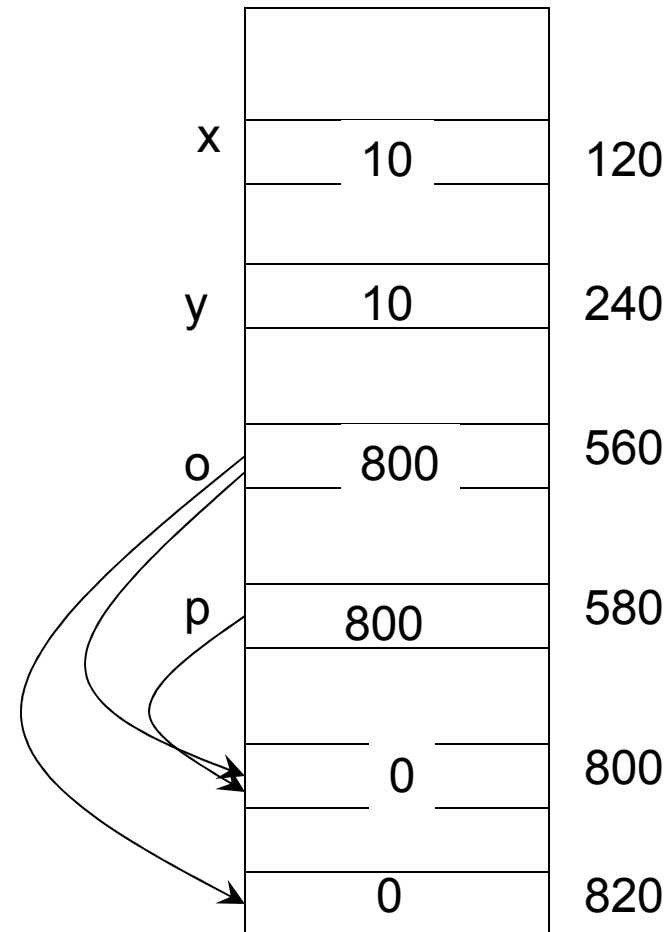
```
public class aClass{
    aClass(int value){
        m=value;
    }
    public int m;
}
[...]
```

```
Public static void main(){
    int x=0, y=10;
    aClass o = new aClass(0);
    aClass p = new aClass(10);
    x=y;
    x=0;
    System.out.println(x+" "+y);
    o=p;
    o.m=0;
    System.out.println(o.m+" "+p.m);
}
```

Output:

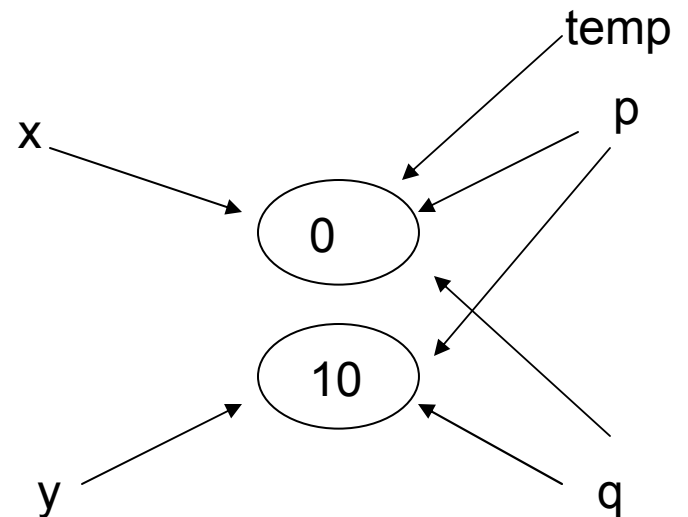
0 10  
0 0

Memory map



```
aClass x = new aClass(0);  
aClass y = new aClass(10);  
swap (x, y);  
System.out.println(x.m+" " +y.m);  
[...]  
void swap (aClass p, aClass q){  
    aClass temp=p;  
    p = q; q = temp;  
}
```

Output: 0 10



---

# Arrays in Java

- To declare an array follow the type with (empty) []s
    - `int[] grade; //or`
    - `int grade[]; //both declare an int array`
  - In Java arrays are objects so must be created with the **new** keyword
    - To create an array of ten integers:
      - `int[] grade = new int[10];`
      - Note that the array size has to be specified, although it can be specified with a variable at run-time
-

---

# Arrays in Java

- ❑ When the array is created memory is reserved for its contents
  - ❑ Initialization lists can be used to specify the initial values of an array, in which case the **new** operator is not used
    - `int[] grade = {87, 93, 35}; //array of 3 ints`
  - To find the length of an array use its `.length` variable
    - ❑ `int numGrades = grade.length; //note: not .length()!!`
  - We can access the elements of an array by indexing
    - ❑ `int x = grade[0];`
-

---

# Object Conversion

- We can convert different types of object to each other.

```
class Fruit { ... }  
class Pineapple extends Fruit { ... }  
public void main () {  
    Pineapple p = new Pineapple();  
    Fruit f=p; //a valid statment  
}
```

The type Fruit can "hold" the type Pineapple since a Pineapple ***is a*** Fruit. Such automatic cases are called conversions.

---

---

# Automatic Conversion rules

```
public class student{
    private int ID;
    public void setID(int ID){this.ID = ID;}
    ...
}

public void main (){
    Object o;
    Student s= new Student();
    o=s; //implicit casting.
    o.setID(3090);
}
```

- ✓ That is, the type Object can "hold" the type Student since a Student ***is an*** Object.
  - ✓ *In general an object of a super-class can be automatically converted to an object of A sub-class.*
-

# Object conversion are useful for implementing ADT's

```
class ListArrayBased {
    private Object items[];
    public void add(int index, Object item){
        ...
    }
}

class Student{
    private int ID;
    Student(int ID){this.ID = ID;}
}

public void main (){
    ListArrayBased listInt = new ListArrayBased();
    ListArrayBased listDouble = new ListArrayBased();
    ListArrayBased listStudent = new ListArrayBased();
    ListInteger.add(0, Integer(-1));
    ListDouble.add(0, Double(3.14));
    Student s = new Student(3090);
    ListStudent.add(0, s);
}
```

**We can create different lists that hold different types of objects.**

---

# Explicit Casting

- An object of a sub-class can also be converted to an object of a super-class if it's explicitly directed.

```
class Fruit { ... }  
class Pineapple extends Fruit { ... }  
public void main () {  
    Fruit f = new Fruit();  
    Pineapple p=f; //illegal conversion  
    Pineapple p=(Pineapple)f; //legal (explicit casting)  
}
```

---

# Explicit Casting (Example)

```
class AClass {  
    void aMethod () { ... }  
}  
  
class BClass extends AClass {  
    void bMethod () { ... }  
}  
  
public void miscMethod (AClass obj) {  
    obj.aMethod ();  
    if (obj instanceof BClass) ((BClass)obj).bMethod ();  
}  
Public void main() {  
    BClass bObj=new BClass();  
    miscMethod(bObj);  
}
```

If you don't explicitly cast obj to a BClass object the compiler produce an error

---

# Interface Conversion

- A class that implements an interface can be **automatically converted** to that interface.

```
interface Sweet { ... }  
class Fruit implements Sweet { ... }  
public void main () {  
    Fruit f=new Fruit();  
    Sweet s;  
    s = f; // legal conversion from class type to interface type  
    f = s; // illegal conversion from interface type to class type  
    f = (Fruit) s; // legal conversion from class type to interface type  
}
```

---

---

# Cast Rules

- There are *compile-time* rules and *runtime* rules.
  - The compile-time rules are there to catch attempted casts in cases that are simply not possible.
    - For instance, classes that are completely unrelated - i.e., neither inherits from the other and neither implements the same interface as the other - cannot be converted to each other.
  - Casts that are permitted at compile-time include
    - casting any object to its own class or to one of its sub or superclass types or interfaces.
    - Almost anything can be cast to almost any interface, and an interface can be cast to almost any class type.
  - The compile-time rules cannot catch every invalid cast attempt. If the compile-time rules permit a cast, then additional, more stringent rules apply at runtime.
  - These runtime rules basically require that the object being cast is compatible with the new type it is being cast to.
-

---

# Comparing objects

- The primitive values can easily be compared.
    - `int x, y, z; if(x < y) , while (z != 0), ...`
  - Object Equality
    - the `Object` class has a method called `equals`
      - Default implementation
        - Compares two objects and returns true if they are actually the same object
      - Customized implementation for a class
        - Can be used to check the values contained in two objects for equality
-

---

# Example

```
class Student{
    private int ID;
    Student(int ID){this.ID = ID;}
}
Public void main(){
    Student s1 = new Student(3090);
    Student s2 = new Student(3090);
    if(s1.equals(s2))
        System.out.println("Equal");
    else
        System.out.println("Not Equals");
    s1=s2;
    if(s1.equals(s2))
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}
```

**Output:**

**Not Equal**

**Equal**

---

---

# Example

```
class Student{
    private int ID;
    Student(int ID){this.ID = ID;}
    boolean equals(Student rhs){
        return ID==rhs.ID;
    }
}

public void main(){
    Student s1 = new Student(3090);
    Student s2 = new Student(3090);
    if(s1.equals(s2))
        System.out.println("Equal");
    else
        System.out.println("Not Equal");
}
```

Output:  
Equal

---

- 
- Class Object does not have a method for comparing the order of the objects.
  - If you need to impose an ordering on the objects use the *Comparable* interface.
  - Comparable interface has a method called `compareTo`.
-

---

```
class Student implements Comparable{
    private int ID;
    Student(int ID){this.ID = ID;}
    int compareTo(Student rhs){
        if(ID==rhs.ID)
            return 0;
        if(ID < rhs.ID)
            return -1;
        return 1;
    }
}

public void main(){
    Student s1 = new Student(3090);
    Student s2 = new Student(2145);
    if(s1.compareTo(s2)<0)
        System.out.println("s1 less than s2");
    else
        System.out.println("s1 greater than or equal to s2");
}
```

---

```
class SortedListArrayBased implements SortedListInterface {

    final int DEFAULT_MAX_SIZE = 1000;
    private Comparable items[];
    private int size, maxSize;
    private int binarySearch(Comparable x){
        [...]
        if (items[middle].compareTo(x)==0)
            return middle; //x is found at the middle location
        [...]
    }
    [...]
}

class Student implements Comparable{
    private int ID;
    Student(int ID){this.ID = ID;}
    int compareTo(Student rhs){
        if(ID==rhs.ID)
            return 0;
        [...]
    }
}

public void main(){

    SortedListArrayBased listInteger = new SortedListArrayBased (100);
    listInteger.insert(new Integer(0));

    SortedListArrayBased listStudents = new SortedListArrayBased (100);
    listStudents.insert(Student(3090));

}
```

# Printing objects.

- The Object class has the member `public String toString()` that converts the Object to a string.
- The `System.out.println()` method automatically calls the `toString` method and prints the returned string on the screen.
- Since every class is either directly or indirectly a subclass of the Object we can call the `toString` method to print any object.

```
class Student{
    private int ID;
    private String name;
    Student(String name, int ID){this.name=name;this.ID = ID;}
    [...]
}
public void main(){
    Student s = new Student("John", 3090);
    System.out.println(s); //it will call the s.toString()
}
```

Output: Student@190d11

This only says that the student object is located at 190d11 in memory

---

# Overriding the toString() method

- The toString method of the Object class can only tell where the object is located in the memory.
- To print the contents of an object we have to override the toString method of the Object class.

```
class Student{
    private int ID;
    private String name;
    Student(String name, int ID){this.name=name;this.ID = ID;}
    public String toString(){
        return "Name:"+name+" ID:"+ID;
    }
    [...]
}

public void main(){
    Student s = new Student("John", 3090);
    System.out.println(s); //it will call the s.toString()

}
```

Output:  
Name:John ID:3090

---

---

# Useful Java Classes

## ■ String classes

### □ Class `String`

#### ■ Declaration examples:

□ `String title;`

□ `String title = "Walls and Mirrors";`

#### ■ Assignment example:

□ `Title = "Walls and Mirrors";`

#### ■ String length example:

□ `title.length();`

#### ■ Referencing a single character

□ `title.charAt(0);`

#### ■ Comparing strings

□ `title.compareTo(string2);`

---

---

# Useful Java Classes

- String classes (continued)

- Class `String`

- Concatenation example:

- ```
String monthName = "December";
```

- ```
int day = 31;
```

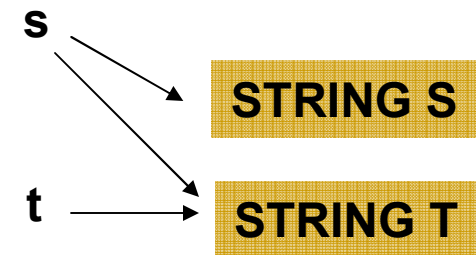
- ```
int year = 02;
```

- ```
String date = monthName + " " + day + ", 20" +  
year;
```

---

# Example

```
public void main(){
    String s = new String("STRING S");
    String t = new String("STRING T");
    System.out.println(s+";"+"t");
    s = t;
    System.out.println(s+";"+"t");
    t = "NEW STRING T";
    System.out.println(s+";"+"t");
}
```



Base on what we learned so far the output must be

```
STRING S ; STRING T
STRING T ; STRING T
NEW STRING T; NEW STRING T
```

But the output is

```
STRING S ; STRING T
STRING T ; STRING T
STRING T; NEW STRING T
```

***The reason is that Strings are immutable***

---

# Mutable vs. Immutable objects.

- **Mutable Objects:** When you have a reference to an instance of an object, the contents of that instance **can** be altered
  - **Immutable Objects:** When you have a reference to an instance of an object, the contents of that instance **cannot** be altered
-

---

```
public void main() {  
    String s = new String("STRING S");  
    String t = new String("STRING T");  
    System.out.println(s+";" +t);  
    s = t;  
    System.out.println(s+";" +t);  
    t = "NEW STRING T";  
    System.out.println(s+";" +t);  
  
}
```

**Question:** If String objects are immutable why the contents of s changes after the statement t="NEW STRING T"; in the above example?

**Answer:** The contents of the object didn't change; we discarded the instance and changed our reference to a new one with new contents.

**Remark:** *It's costly to assign immutable objects.*

*Use the mutable objects when you know the contents will not change or will only change slightly.*

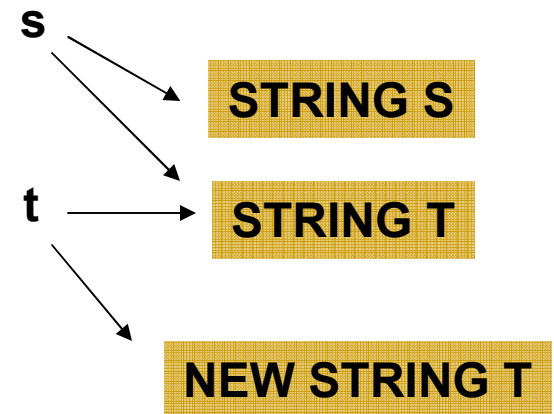
---

# Example

```
public void main(){
    String s = new String("STRING S");
    String t = new String("STRING T");
    System.out.println(s+";"+"t");
    s = t;
    System.out.println(s+";"+"t");
    t = "NEW STRING T";
    System.out.println(s+";"+"t");
}
```

But the output is

```
STRING S ; STRING T
STRING T ; STRING T
STRING T; NEW STRING T
```



---

# Useful Java Classes

## ■ Class `StringBuffer`

- Creates mutable strings
  - Provides same functionality as class `String`
  - More useful methods
    - **public** `StringBuffer append(String str)`
    - **public** `StringBuffer insert(int offset, String str)`
    - **public** `StringBuffer delete(int start, int end)`
    - **public** `void setCharAt(int index, char ch)`
    - **public** `StringBuffer replace(int start, int end, String str)`
-

---

## Which classes are Immutable?

- All of the java.lang package wrapper classes are immutable: Boolean, Byte, Character, Double, Float, Integer, Long, Short, String.
  - The class that you define are mutable.
-