

1. Conversions and Overloading

The determination of type correctness is actually not as simple as described previously, for two reasons.

First. Java allows certain implicit conversions of a value of one type to a value of another type.

Implicit conversations involve only **the primitive types**. For example, Java allows **chars** to be **widened** to the numeric types. Thus, the assignment to **n** in the following is legal:

```
char c='A';  
int n=c;
```

Second, conversations involve computation, that is, they cause a production of a new type (of the variable types) that is then assigned to the variable.

After the compiler determines the conversation needed to make the assignment legal, it generates the code needed to produce the new value.

2. Overloading

In addition, Java allows **overloading**. This means that there can be several methods with **the same name**.

For example, consider a class C with the following methods:

```
Public class c {  
static int comp( int i, long j) // def1.  
static float comp( long i, int j) // def 2  
static int comp (long i, long j) // def3  
}
```

This class provides three overloaded definitions of *comp*. when there are overloaded definitions, several of them might work for a particular call. For example, suppose you have the declarations:

```
int x;  
long y;  
float z;
```

In Java, an *int* (32 bits) can be widened to a *long* (64 bits), and also a *float* (32 bits) can be widened to a *long* (64 bits), Therefore a call C.com(x, y) could go to either the first definition of comp (**since here the types match exactly**) or the third definition of comp (by widening x to a long). The second definition is not possible since it isn't possible to widen a long to an int.

The rule used to determine which method to call when there are several choices, as in this example, is “**most specific**”. A method m1 is more specific than another method m2 if any legal call of m1 would also be a legal call of m2 if more conversations were done.

For example, the first definition of comp would be selected for the call C.com(x,y) since it is more specific than the third definition.

Note:

If there is no most specific method, a compile time error occurs for example, all three definitions matches for the call C.comp(x,x). however, none of these is most specific and therefore the call is illegal .

The programmer can solve the ambiguity in case like this by making the conversation explicitly, for example **C.comp ((long) x,x)** select the second definition.

2.1. Constructor Overloading

As with methods, constructors can be overloaded.

An example is:

Public class Employee {

String name;

```
public Employee(String name, double salary, Date
DoB) .
```

```
{
This.name=name;// this.name is different from the
this , where this is used to call the overloading
constructor.
```

```
this(name; null);
```

```
}
```

```
public Employee(String name, double salary).
```

```
{
```

```
this(name, 0.12, null)
```

```
}
```

```
public Employee(String name, Date DoB)
```

```
}
```

- Argument lists *must* differ.
- You can use the `this` reference at the first line of a constructor to call another constructor.

```
public class Employee {
    private static final double BASE_SALARY = 15000.00;
    private String name;
    private double salary;
    private Date birthDate;

    public Employee (String name, double salary, Date
DoB) {
        this.name = name;
        this.salary = salary;
        this.birthDate = DoB;
    }
    public Employee (String name, double salary) {
        this(name, salary, null);
    }
    public Employee (String name, Date DoB) {
        this(name, BASE_SALARY, DoB);
    }
    // more Employee code...
}
```

2.2. Constructing and Initializing Objects:

Memory is allocated and default initialization occurs.

Instance variable initialization uses these steps recursively:

1. Bind constructor parameters.
2. If explicit this(), call recursively, and then skip to Step 5.
3. Call recursively the implicit or explicit super call, except for Object.
4. Execute the explicit instance variable initializers.
5. Execute the body of the current constructor.

Constructor and Initialization Examples

```
public class Object {  
    public Object() // constructor for object clasds{}  
}
```

```
public class Employee extends Object {
```

```
    private String name;  
    private double salary = 15000.00;  
    private Date birthDate;
```

```
    public Employee(String n, Date DoB) {  
        // implicit super();  
        this.name = n;  
        this.birthDate = DoB;  
    }
```

```
    public Employee(String n) {  
        this(n, null);  
    }  
}
```

```
public class Manager extends Employee {  
    private String department;  
    public Manager(String n, String d) {  
        super(n);  
        department = d;  
    }  
}
```

Constructor and Initialization Examples

- 0 Basic initialization
 - 0.1 Allocate memory for the complete Manager object
 - 0.2 Initialize all instance variables to their default values (0 or null)
- 1 Call constructor: Manager("Joe Smith", "Sales")
 - 1.1 Bind constructor parameters: n="Joe Smith", d="Sales"
 - 1.2 No explicit this() call
 - 1.3 Call super(n) for Employee(String)
 - 1.3.1 Bind constructor parameters: n="Joe Smith"
 - 1.3.2 Call this(n, null) for Employee(String, Date)
 - 1.3.2.1 Bind constructor parameters: n="Joe Smith", DoB=null
 - 1.3.2.2 No explicit this() call
 - 1.3.2.3 Call super() for Object()
 - 1.3.2.3.1 No binding necessary
 - 1.3.2.3.2 No this() call
 - 1.3.2.3.3 No super() call (Object is the root)
 - 1.3.2.3.4 No explicit variable initialization for Object
 - 1.3.2.3.5 No method body to call