Advanced Programming Techniques

Session 9 : 04-Feb-2018

This PPT covers the module M9 from the course handout

Agenda

- Java Types & Type Inquiry
- Object Class
- Shallow & Deep Copy
- Serialization
- Reflection

Types Definition

- What is type in Java?
  - ➔Set of Values with a set of Operations that can be applied to the values.

Example :

int type -→ Set of values { [-2^31] to [+2^31-1])}

Operations : All Arithmetic Operations

BOX type -→ Set of all BOX object references

Operations : area() , volume() , toString()

- Java is Strongly Typed Language.
- Type checking is done both at Compile and Run Time.
- If a Type Check Fails at Compile Time -→ Compile Time Error
- If a Type Check Fails at Run Time -→ Run Time Exception

Types in Java

1. *Primitive Type(int, float, double, byte, char, boolean, short, long)*
2. *A class Type [BOX, String, Arrays , Student etc]*
3. *An interface Type [Either User Defined OR Libarary interfaces such as Comparable]*
4. *An array type*
5. *null type*

Note :

1. *Arrays have a component type (Type of array elements) e.g String[] array has component type as String; int[] array has Component type as int*
2. *void is not a type in java. [only a keyword to denote that a method does not return anything]*
3. *Object[] names = new String[10]; What's Array Type and its component type.*

Values in Java

1. A Value of Primitive type
2. A reference to an object
3. reference to an array
4. Null type is a subset of all in Java

Examples :

10 , 10.35 , true → Primitive Type Values

new BOX(10,6,8); → Reference to an object

new int[] { 10,6,8} → Reference to array

null

Note : You can not have a value of type interface. ➔ WHY ??

Nonprimitive types

	type	object
class types	`Rectangle` `String`	`new Rectangle(2,4,8,8)` `"dProg2"`
interface types	`Shape` `Comparable`	
array types	`int[][]` `String[]`	`new int[3][7]` `{"dIntProg","dProg2"}`

- no objects of interface type!

type	value
null type	`null`

Null vs Void

It is not really meaningful to speak about the difference between **null** and **void**, because they are two separate concepts:

- When a variable has the value **null**, it means that the variable isn't referring to any object.
- A **void** method is a method that doesn't return a value.

So **null** is a specific value that a variable can have, and **void** indicates that a method has a certain property - two completely different things.

A variable takes up space in memory, no matter if it refers to an object or not.

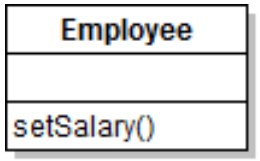
The keyword **void** is only used to indicate that a method doesn't return a value. The keyword **void** doesn't make any sense anywhere else in Java source code.

QUIZ – Type Checking

static/dynamic type

	All Good	Compile Error	Runtime Error
1.	a),b)		
2.	a)	b)	
3.	a)		b)
4.	b)	a)	
5.		a),b)	
6.		a)	b)
7.	b)		a)
8.		b)	a)
9.			a),b)

10. I don't know

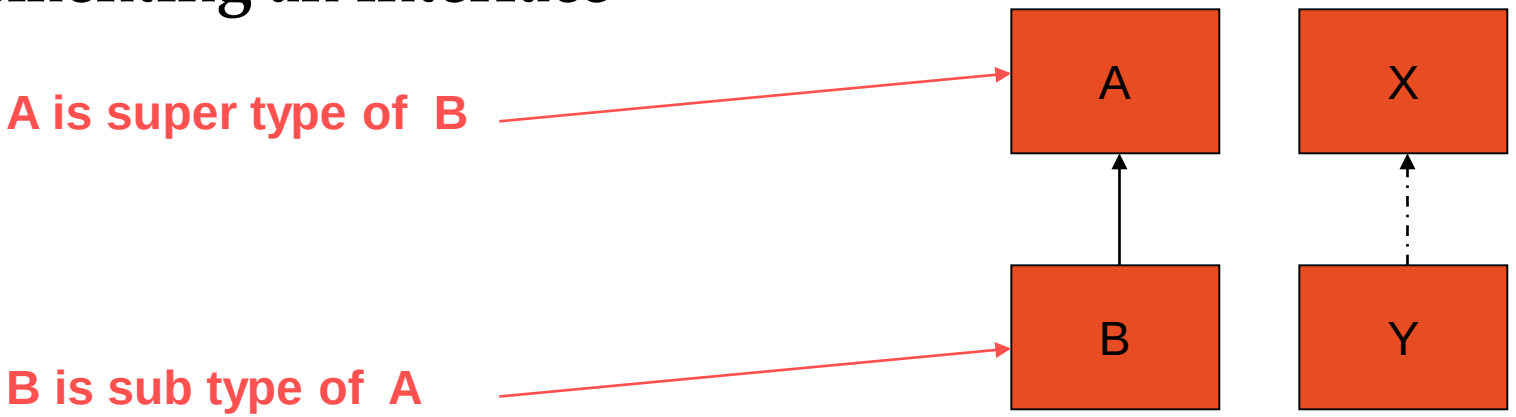



```
Employee e = null;

a) e.clear();
b) e.setSalary(1000);
```

Sub Types

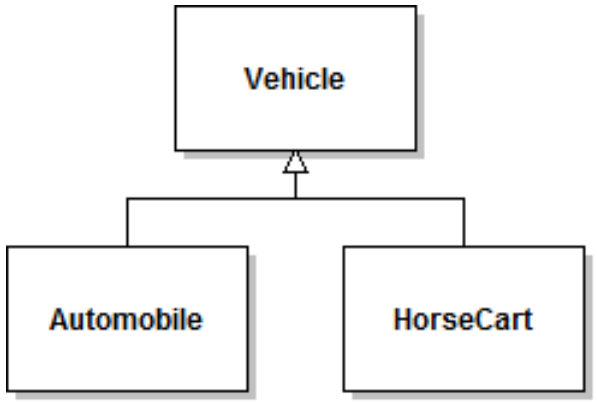
- **Sub type specifies the inheritance relationship either by extending a class or implementing an interface**



Similarly X is super type for Y and Y is sub type for X.

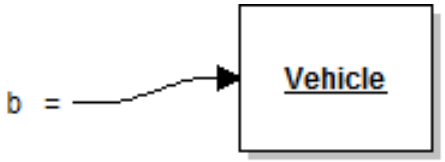
You can substitute a value of subtype whenever supertype value is expected

Non-primitive type: variables



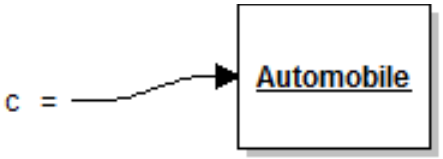
- Variables of non-primitive type contains reference to object of **same** type

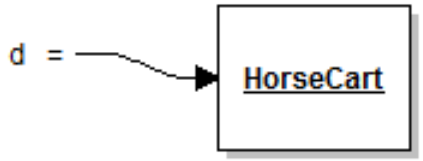
```
Vehicle b;
  b = new Vehicle();
```



- or of a **sub**type

```
Vehicle c;
  c = New Automobile();
Vehicle d;
  d = new HorseCart();
```





Example

What's Expected

X x1 = ?

x1 is reference variable of type X?

If X is an interface

RHS can be an instance of any class implementing X.

If X is abstract class

RHS can be an instance of any concrete subclass of X

If X is a concrete class

RHS can be either an instance of X or any of its subclasses

Rules for Subtype Relationships

S is a subtype of T if

S and T are the **same type**

S and T are **both class types**, and T is a direct or indirect **superclass** of S

S is a **class type**, T is an **interface type**, and S or one of its super-classes **implements** T

S and T are **both interface types**, and **T is a direct or indirect super-interface of S**

S and T are both **array types**, and the **component type of S** is a **subtype of the component type** of T

S is **not a primitive type** and T is the type Object

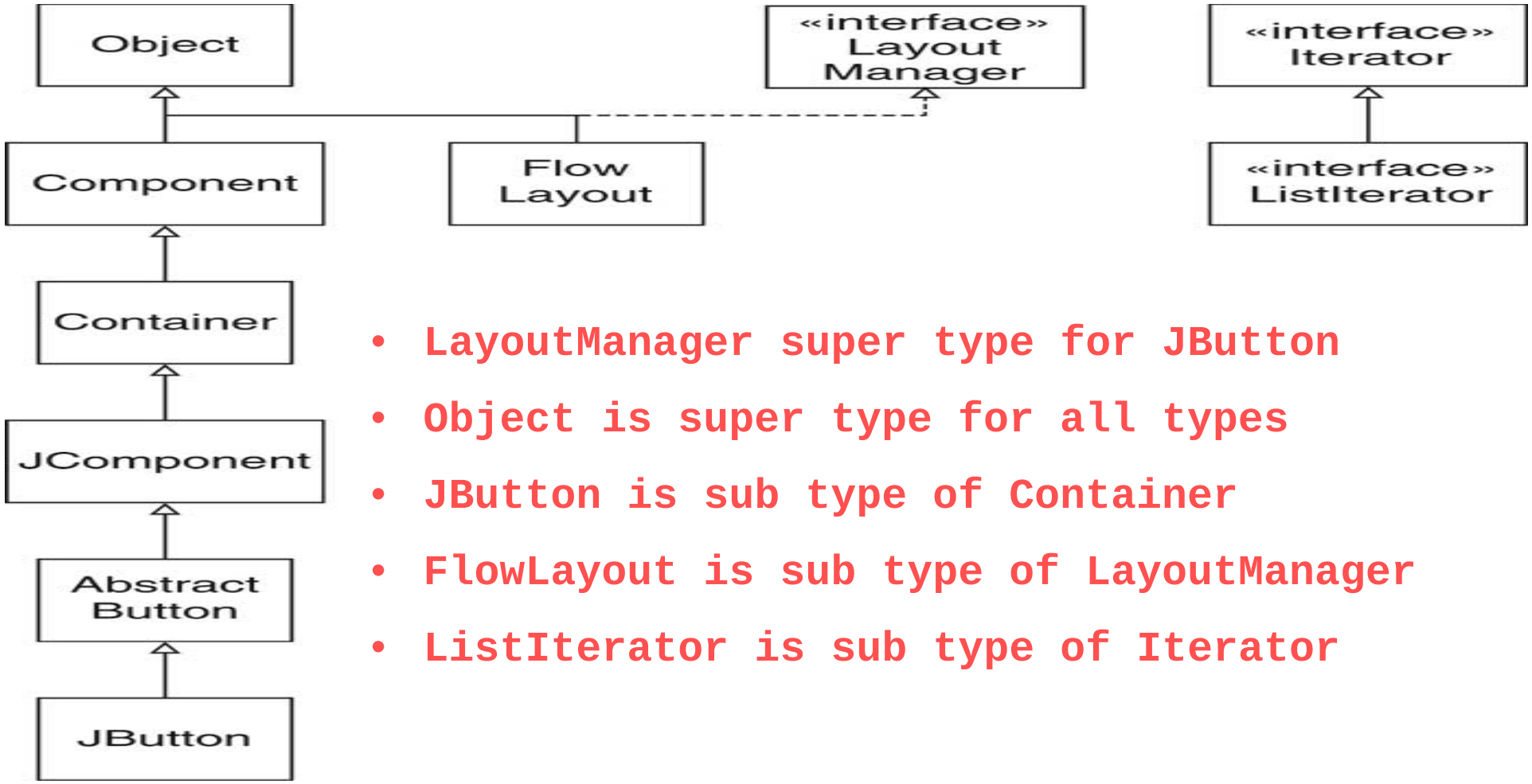
S is an array type and T is Cloneable or Serializable

S is the null type and T is not a primitive type

Subtype Relationship

- T1 is subtype of T2, T1 T2
  - if T1 is the same type as T2
  - or T1 implements T2
  - or T1 extends T2
  - or there is T3 such that T1 T3 and T3 T2
  - or T1 is T3[], T2 is T4[] and T3 T4
  - or T1 is array type and T2 is Cloneable or Serializable
  - or T1 is non-primitive and T2 is Object
  - or T1 is null type and T2 is non-primitive

Examples



- **LayoutManager super type for JButton**
- **Object is super type for all types**
- **JButton is sub type of Container**
- **FlowLayout is sub type of LayoutManager**
- **ListIterator is sub type of Iterator**

The ArrayStoreException

- This exception is thrown when there has been made an attempt to store the wrong type of object into an array of objects.

- The ArrayStoreException extends RuntimeException

How to deal with ArrayStoreException

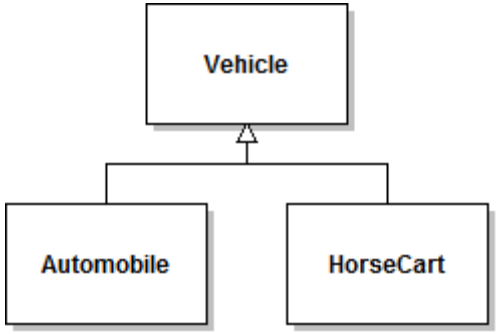
Whenever you see this exception, it means that you have been storing a wrong kind of data type in an array.

One thing that may solve this, is the usage of the proper type, or even casting to the proper type.

A way to prevent this exception, is to use a less generic data type in your arrays.

If the above example fits, it would be a good idea not to use Object as the array type, but maybe Integer or String, depending to the use case.

Type check: compile time



- Static versus dynamic type

```
Vehicle v;
  v = new Automobile();
Object obj;
  obj = v;
```

Static type of v is `Vehicle`

Dynamic type of v is `Automobile`

- Compiler looks at **static** type only

```
Automobile bmw
  bmw = v;
```

compile time error

- Use **type cast**

```
  bmw = (Automobile) v;
```

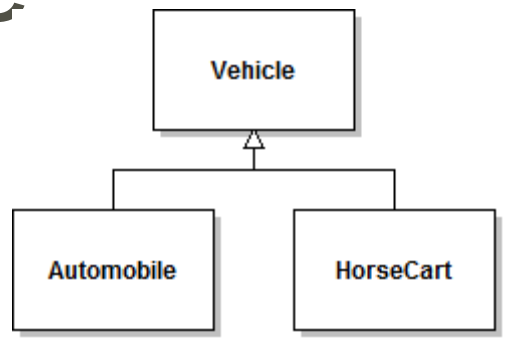
Static type of `(Automobile)v` is `Automobile`

- Compiler is happy!

Type check: runtime

- You may fool the compiler:

```
Vehicle v = new HorseCart();
Automobile bmw = (Automobile) v;
```



Static type of `(Automobile)v` is `Automobile`

- Compiler is happy
- But at runtime

Exception in thread "main“
java.lang.ClassCastException: `HorseCart` cannot be cast to `Automobile`

Dynamic type of `v` is `HorseCart`

Wrapper Classes

1. Primitive types aren't Objects in Java.
2. Primitive wrapper classes are not the same thing as are primitive types. The main difference is that whereas variables can be declared in Java as **double**, **short**, **int**, or **char**, etc., data types, the eight primitive wrapper classes create instantiated objects and methods that inherit but hide the eight primitive data types, not variables that are assigned data type values
3. We Can wrap primitive types in Objects using wrapper classes.

Primitive type	Wrapper class	Constructor Arguments
byte	Byte	byte or String
short	Short	short or String
int	Integer	int or String
long	Long	long or String
float	Float	float, double or String
double	Double	double or String
char	Character	char
boolean	Boolean	boolean or String

Source : Wikipedia

Wrapper Classes

Generally, you should use primitive types unless you *need* an object for some reason (e.g. to put in a collection). Even then, consider a different approach that doesn't require a object if you want to maximize numeric performance.

Performance Issues

A few experiments have shown that using wrapper can be inefficient, and can give you a false sense of efficiency. What may look like an efficient use of primitive data types at the source-code level could well turn out to be a very inefficient use of primitive data wrapper types when it comes to the runtime.

Wrapper classes are immutable.

Boxing / Un-Boxing

Boxing is a way to wrap primitive types with objects so that they can be used like objects.

For example,

theInteger class wraps the int primitive type in Java.

Similarly, ***unboxing*** is a way to convert object types back to primitive types.

From the programmer's perspective, reducing code size and improving performance is an challenging job.

The need to explicitly convert data from primitives to object references arises frequently and is often burdensome.

Another annoying problem with using traditional casting techniques is that it clutters up the code.

```
class BoxingExample1{
 public static void main(String args[]){
  int a=50;
     Integer a2=new Integer(a);//Boxing

     Integer a3=5;//Boxing

     System.out.println(a2+" "+a3);
 }
}
```

```
class UnboxingExample1{
  public static void main(String args[]){
    Integer i=new Integer(50);
      int a=i;

      System.out.println(a);
  }
 }
```

Parametric Types

ArrayList countries = new ArrayList();

Countries.add(new Country(...));

Countries.add("France");

- No compiler error.
- No run time error as the add method parameter type is object.

void add(Object element)

- If you try to retrieve country object

Country c = (Country) countries.get(i);

- This throws run time error as the content is a string,
- Use Template for Parametric Type:

ArrayList<Country> = new ArrayList<Country> ();

- The add method will now have an add method:

void add(Country element)

Enumerated Types

1. Type with finite set of values
2. Example: enum Size { SMALL, MEDIUM, LARGE }
   public enum MartialStatus { MARRIED, UNMARRIED}

3. Typical use:
   Size imageSize = Size.MEDIUM;
   if (imageSize == Size.SMALL) . . .
4. Syntax :
   accessSpecifier enum TypeName { value1, value2, ……value n}
5. Safer than integer constants
   public static final int SMALL = 1;
   public static final int MEDIUM = 2;
   public static final int LARGE = 3;

Type safe Enumerations

1. enum equivalent to class with fixed number of instances

```
public class Size
{
   private /* ! */ Size() {  }
   public static final Size SMALL = new Size();
   public static final Size MEDIUM = new Size();
   public static final Size LARGE = new Size();
}
```

2. enum types are classes; can add methods, fields, constructors

Points to Remember

Points to remember for Java Enum

enum improves type safety

enum can be easily used in switch

enum can be traversed

enum can have fields, constructors and methods

enum may implement many interfaces but cannot extend any class

because it internally extends Enum class

Type Inquiry

- Test whether e is a Shape:
  if (e instanceof Shape) . . .
- Common before casts:
  Shape s = (Shape) e;
- Don't know exact type of e
- Could be any class implementing Shape
- If e is null, test returns false (no exception)

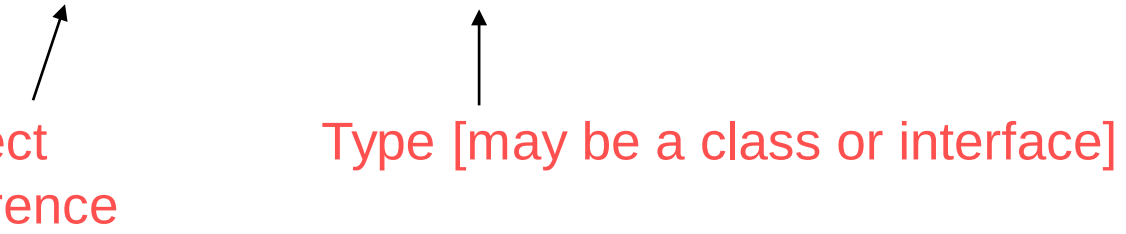
The ***instanceof*** operator tests whether the type of an object is a subtype of the given type

- **Instanceof** testing is often done before we apply a cast to make sure the cast does not fail.
- Instanceof a class is true
  - The object tested may belong to the class or any of its subclasses.

Type Inquiry

1. instanceof operator tests whether the type of an object reference is a subtype of given type or not.
2. Syntax :

   if(e instanceof S) { ..... }



3. The above statement tests whether e is a instance of type S or not.
4. This statement returns true if e is a direct instance of S or e belongs to one of the sub classes of S.
5. Instanceof operator can test whether the type of a value is subtype of a given type or not. But it does not give the exact type to which e belongs.
6. If e is null then instanceof does not throw an Exception but simply returns false.

Class class

- The **java.lang.Class** class instance represent classes and interfaces in a running Java application.

- **getClass** method gets class of any object
- Returns object of **type Class**
- Class object describes a ***type***

```
Object e = new Rectangle();
Class c = e.getClass();
System.out.println(c.getName());
                        // prints java.awt.Rectangle
```

- Class.forName method yields Class object:

```
Class c = Class.forName("java.awt.Rectangle");
```

- .class suffix yields Class object:

```
Class c = Rectangle.class;
                        // java.awt prefix not needed
```

An object of of the **Class** class is a descriptor for a type.

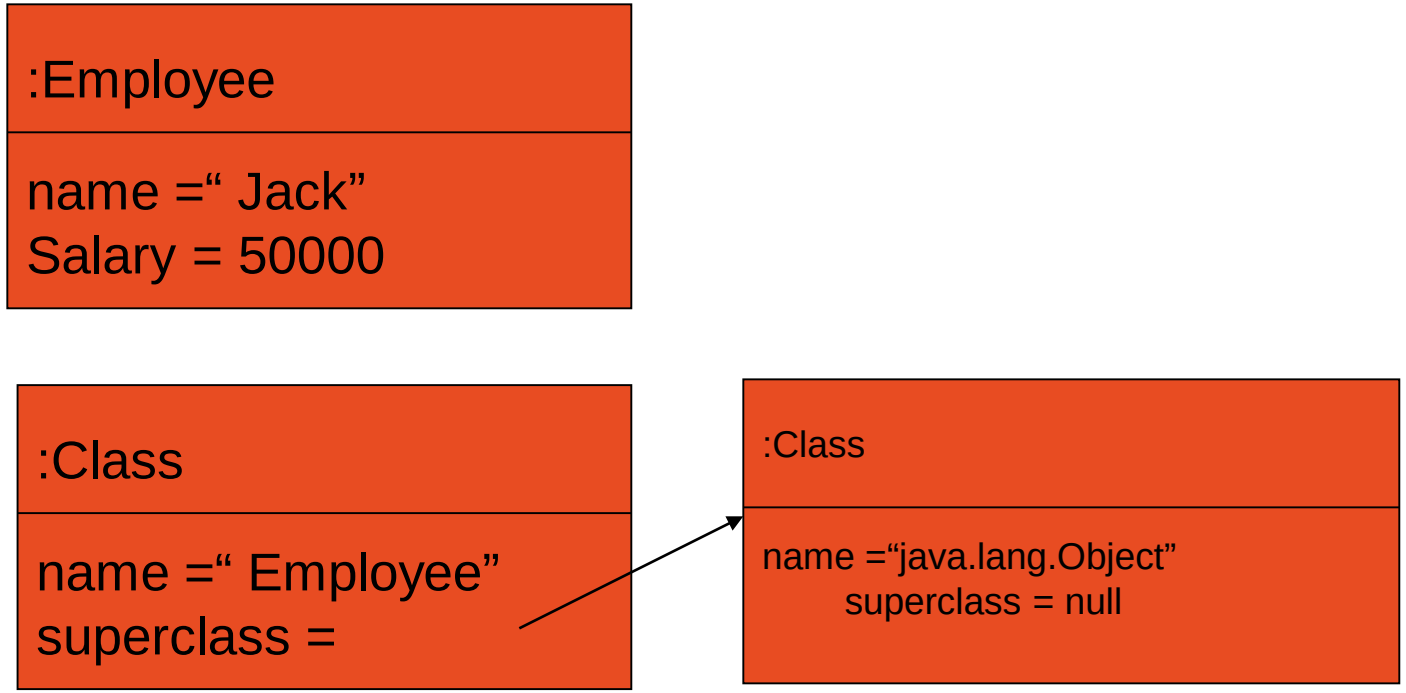
Class is a misnomer:

Class objects can describe any type, including primitive types, class types, and interface types. Eg.:

int.class,
void.class,
Shape.class

Class class

- A Class object is a type descriptor. It contains information about a given type such as type name and super class.



Class class

- Given a object reference we can know the exact type of the object by using getClass() method

```
Class c = e.getClass();
```

- getClass() method returns a Class Object. Once you have a class object its name can be printed as follows

```
System.out.println(e.getClass().getName());
```

- Adding the suffix .class to a type also yields the Class object.

Knowing Exact class of Reference

1. Adding a suffix .class to a class name always yields a Class Object.
2. To test whether std is a reference belonging to class Student or not use

   if (std.getClass() == Student.class)

3. To test whether emp is a refrence for Employee class object or not

   if(emp.getClass() == Emplyoee.class)

4. What about Arrays ?

```
BOX[ ] box = new BOX[5];
Class c = box.getClass();
if( c.isArray())
  S.O.P (" Component Type :"+ c.getComponentType());
```

Object class

The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know.

Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object.
For example:

```
Object obj=getObject();//we don't know what object will be returned from this method
```

The Object class provides some common behaviors to all the objects such as **object can be compared**, **object can be cloned**, **object can be notified** etc.

Object class

- Common super class for all other java classes.
- A class which is defined without extends clause is a direct sub class of Object class.
- Methods of Object class applies to all Java Objects.
- Important Methods:

1. String toString() ➔ Useful for debugging
2. boolean equals(Object other) ➔ Comparison
3. int hashCode()
4. Object clone()

public String toString()

- Returns a string representation of the object
- Useful for debugging
- toString used by concatenation operator
- aString + anObject means aString + anObject.toString()
- User can override the toString() method.

```
class BOX
{
……
public String toString()
{
……
……..
}
}
```

```
class Student
{
……
public String toString()
{
……
……..
}
}
```

toString() continued…

- toString() is automatically called when you
  1. concatenate an object with a string
  2. print an object with print or println method
  3. when you pass an object reference e to assert statement

- Default implementation of toString() method returns the name of the class and the hash code of the object.
- Can be overridden, more often than not as it provides useful debug information
- Usage from sub Class is also possible. In this case, the tostring will output details of the parent plus any additional details from the child as per implementation

.equals Comparison

- equals method tests whether two objects have equal contents or not.
- Equals method must be *reflexive*, *symmetric* and *transitive*.
- x.equals(x) should return true. (Reflexive)
- x.equals(y) returns true iff y.equals(x) returns true
- If x.equals(y) returns true and y.equals(z) returns true then x.equals(z) should also return true.
- For any non-null reference, x.equals(null) should return false.
- Users can either overload or override equals() method.

.equals Overriding

- Notion of equality depends on class
- Common definition: compare all fields

```
public class Employee
{
  public boolean equals(Object otherObject)
  // not complete--see below
  {
    Employee other = (Employee)otherObject;
      return name.equals(other.name)
      && salary == other.salary;
   }
      ...
}
```

- Must cast the Object parameter to subclass
- Use == for primitive types, equals for object fields

Overriding equals in Subclass

- Call equals on superclass

```
public class Manager
{
   public boolean equals(Object
     otherObject)
   {
     Manager other =
     (Manager)otherObject;
     return super.equals(other)
     &&
     department.equals(other.departme
     nt);
   }
}
```

.equals (cont)

- Two sets are equal if they have the same elements *in some order*

```
public boolean equals(Object o)
{
    if (o == this) return true;
    if (!(o instanceof Set)) return false;
    Collection c = (Collection) o;
    if (c.size() != size()) return false;
    return containsAll(c);
}
```

reflexive: x.equals(x)
symmetric: x.equals(y)
if and only if y.equals(x)
transitive: if x.equals(y) and y.equals(z),
then x.equals(z)
x.equals(null) must return false

The Object.equalsMethod

- Object.equals tests for identity:

```
public class Object
{
    public boolean
    equals(Object obj)
    {
        return this == obj;
    }
    …
}
```

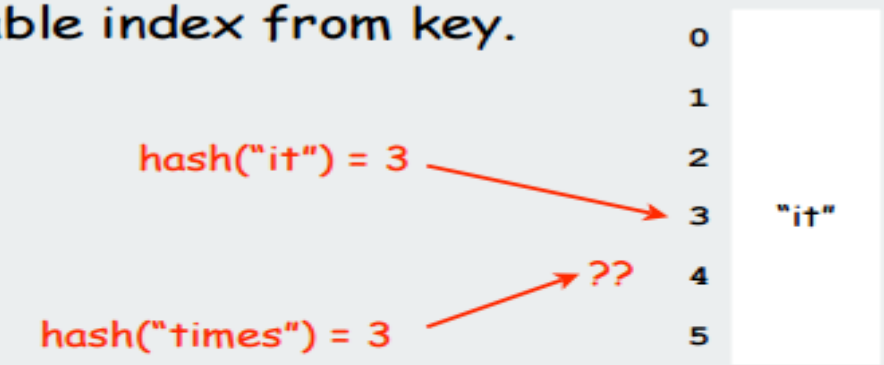
- Override equals if you don't want to inherit that behavior

Hashing

If two objects are equal, then calling hashCode() on both objects must return the same value.

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing table index from key.



Issues.

1. Computing the hash function
2. Collision resolution: Algorithm and data structure to handle two keys that hash to the same index.
3. Equality test: Method for checking whether two keys are equal.

Hashing

What would happen if the two objects are equal but return different hashCodes?

Your code would run perfectly fine. You will never come in trouble unless and until you have not stored your object in a collection like HashSet or HashMap.

But when you do that, you might get strange problems at runtime.

To understand this better, you have to first understand how collection classes such as HashMap and HashSet work.

These collections classes depend on the fact that the objects that you put as a key in them must obey the above contract.

You will get strange and unpredictable results at runtime if you do not obey the contract and try to store them in a collection.

Hashing

Java's `hashCode()` convention

Theoretical advantages

- Ensures hashing can be used for every type of object
- Allows expert implementations suited to each type

Requirements:

- If `x.equals(y)` then `x` and `y` must have the same hash code.
- Repeated calls to `x.hashCode()` must return the same value.

Practical realities

- True randomness is hard to achieve
- Cost is an important consideration

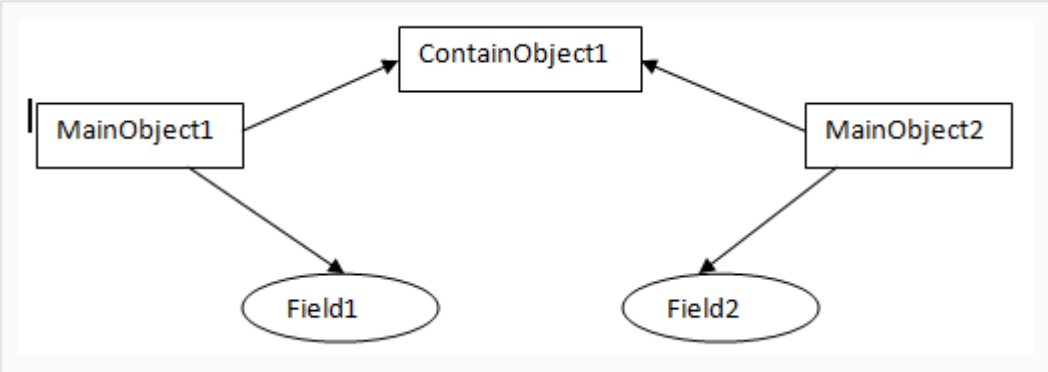
Available implementations

- default (inherited from Object): Memory address of `x` (!!!)
- customized Java implementations: `String`, `URL`, `Integer`, `Date`.
- User-defined types: users are on their own

that's you!

Shallow and Deep copy

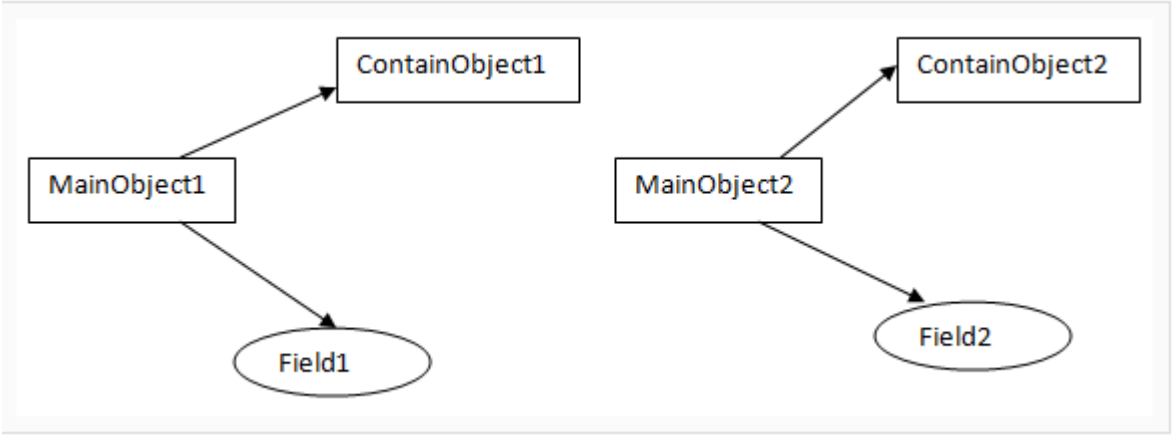
Shallow copy is a bit-wise copy of an object. A new object is created that has an exact copy of the values in the original object. If any of the fields of the object are references to other objects, just the reference addresses are copied i.e., only the memory address is copied.



In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a shallow copy of MainObject1, MainObject2 is created with "field2" containing the copied value of "field1" and still pointing to ContainObject1 itself. Observe here and you will find that since field1 is of primitive type, the values of it are copied to field2 but ContainedObject1 is an object, so MainObject2 is still pointing to ContainObject1. So any changes made to ContainObject1 in MainObject1 will reflect in MainObject2.

Shallow and Deep copy

A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

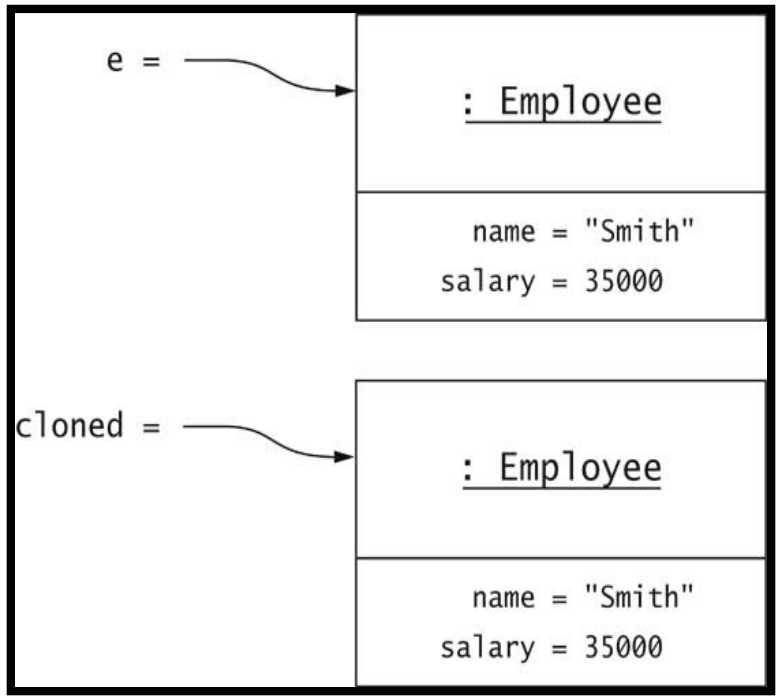


In this figure, the MainObject1 have fields "field1" of int type, and "ContainObject1" of ContainObject type. When you do a deep copy of MainObject1, MainObject2 is created with "field2" containing the copied value of "field1" and "ContainObject2" containing the copied value of ContainObject1.So any changes made to ContainObject1 in **MainObject1 will not reflect** in MainObject2.

Cloning and Clonable Interface

- **clone() method is used to make the clone or deep of the object.**
- **Example :**

```
Employee e = new Employee(…..);
Employee cloned = (Employee) e.clone();
```




```
Assumption :
Employee class supplies a suitable clone() method
```

Cloning Conditions

- x.clone() != x
- x.clone().equals(x) return true
- x.clone().getClass() == x.getClass()

“ clone should be a new object but it should be equals to its original”

Cloning Summary

- Any class willing to be cloned must

1. Declare the clone() method to be public
2. Implement Cloneable interface

class Employee *implements* ***Cloneable***

{

public Object ***clone()***

{

try { super.clone() }

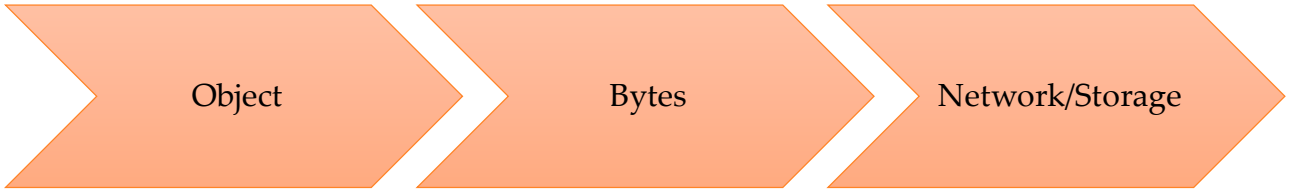
catch(CloneNotSupportedException e){ .. }

}

Serialization

… is the process of translating data structures or object state into a format that can be stored (for example, in a file or memory buffer, or transmitted across a network connection link) and resurrected later in the same or another computer environment.

-- WIkipedia



Use-cases

- **Transfer data over network** between cluster nodes or between servers and clients
  - **Serialization** is a key technology behind any RPC/RMI
- Serialization if often used for **storage**
  - but then data transfer is still often a part of the picture

Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes ObjectInputStream and ObjectOutputStream are high-level streams that contain the methods for serializing and deserializing an object.

Serialization

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out –

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object –

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

Serialization

Reflection denotes the ability of a program to analyse the objects and their capabilities at run time.

Reflection Class	Purpose
Class	Describes a type
Package	Describes a package
Field	Describes a field and allows inspection and modification of fields
Method	Describes a method and allows invocation on objects
Constructor	Describes a constructor and allows its invocation
Array	Has static methods to analyse arrays

Just as the Class class can be demystified by thinking of it as a type descriptor you could think of other Reflection classes as descriptor. An Object of Method Class: Method Object is not a method just like a Object of Class class is not a Class; It only describes the method name, its parameters and return type.

End of Session