**COPYING OBJECTS**

In other languages (common in C++), to enable clients to easily make copies of an object, you can supply a copy constructor :

// in client code

Point p1 = new Point(-3, 5);

Point p2 = new Point(p1); // make p2 a copy of p1

// in Point.java

public Point(Point blueprint) { // copy constructor

this.x = blueprint.x;

this.y = blueprint.y;

}

JAVA has some copy constructors (Actually there is no term as copy constructor in JAVA. They are not special constructors. In C++ they are special because even if you don’t use them, compiler uses them to make copies of objects when you return objects from the functions or when you send copies of objects to functions by call-by-value. We don’t have such things in JAVA. We don’t send our objects to methods by call-by-value. We only send our objects through references. We send our object references to the method which is call-by-value.) but also has a different way (clone method).

**Object clone Method**

By default, clone method is protected. If you want to make your classes cloneable, you need to override clone method and make it public.

Returns reference of the copy object.

protected Object clone()

//by default clone is protected in class Object

//since it is protected, you cannot call it from the outside

throws CloneNotSupportedException

//throws this when it is called bc you should override it

// if you want your object to be cloneable

– Creates and returns a copy of this object. After you override clone method, it should satisfy these rules:

* x.clone() != x 🡪 reference check so they are not equal
* x.clone().equals(x)
* x.clone().getClass() == x.getClass()
  + All three are true (though none of the above are absolute requirements)

– The Object class's clone method makes a "shallow copy" of the object, but by convention, the object returned by this method should be independent of this object (which is being cloned).

**Protected Access**

protected: Visible only to the class itself, its subclasses, and any other classes in the same package.

– In other words, for most classes you are not allowed to call clone .

– If you want to enable cloning, you must override clone .

• You should make it public so clients can call it.

• You can also change the return type to your class's type. (good)

• You can also not throw the exception. (good)

– You must also make your class implement the Cloneable interface to signify that it is allowed to be cloned. To make your classes cloneable, when you are overriding clone method, you need to implement Cloneable interface. Inside that interface, there is nothing (tagging interface). You are saying to compiler that I can be cloned. Compiler will test your objects through this interface: “Is this an object of Cloneable?” (Recall instanceof method).

a instanceof Cloneable 🡪 should return true if a is a cloneable object

**The Cloneable Interface 🡪 public interface Clonable{ }**

Why would there ever be an interface with no methods?

– Another example: Set interface, a sub-interface of Collection

• **tagging interface**: One that does not contain/add any methods, but is meant to mark a class as having a certain quality or ability.

– Generally a wart in the Java language; a misuse of interfaces.

– Now largely unnecessary thanks to annotations (seen later).

– But we still must interact with a few tagging interfaces, like this one.

**Clone For Point Class**

Flawed Clone Method

public class Point implements Cloneable {

private int x, y;

...

public Point clone() {

Point copy = new Point(this.x, this.y);

return copy;

}

}

What is wrong?

//also implements Cloneable and inherits clone()

public class Point3D extends Point {

private int z;

...

}

Point3D class’s clone method produces a Point! This is undesirable.

The only way to ensure that the clone will have exactly the same type as the original object (even in the presence of inheritance) is to call the clone method from class Object with super.clone()

Proper Clone Method

public class Point implements Cloneable {

private int x, y;

...

public Point clone() {

try{

Point copy = (Point) super.clone();

//gives me the Point object which is shallow copied

return copy;

}

catch (CloneNotSupportedException e){

//this will never happen

return null;

}

}

}

To call Object's clone method, you must use try/catch.

* But if you implement Cloneable, the exception will not be thrown.

Flawed Clone Method 2

public class BankAccount implements Cloneable {

private String name;

private List<String> transactions;

...

public BankAccount clone() {

try {

BankAccount copy = (BankAccount) super.clone();

return copy;

}

catch (CloneNotSupportedException e) {

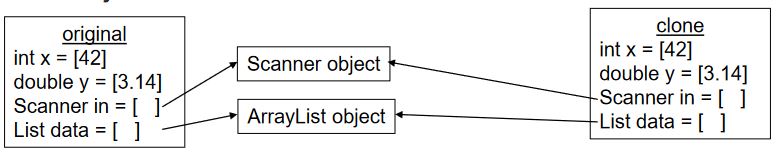
return null; // won't ever happen

}

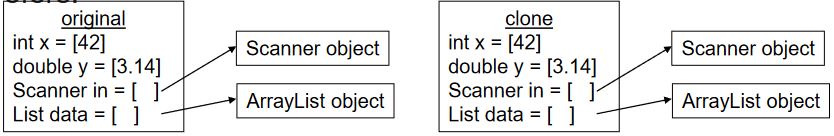
}

}

What is wrong?

Shallow Copy : Duplicates an object without duplicating any other objects to which it refers.

Deep Copy : Duplicates an object's entire *reference* graph: copies itself and deep copies any other objects to which it refers.



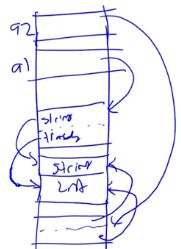
Object’s clone method makes a shallow copy by default.

But if you do “BankAccount a1 = new BankAccount(...);” You got:

Then if you make a shallow copy: “BankAccount a2 = a1.clone();”

This is not good bc if you make a change in one of the lists, both of them will see the effect. So you need deep copying.

I don’t have problem with string bc it is inmutable.



Proper Clone Method 2

public class BankAccount implements Cloneable {

private String name;

private List<String> transactions;

...

public BankAccount clone() {

try { // deep copy

BankAccount copy = (BankAccount) super.clone();

copy.transactions= new ArrayList<String>(transactions);

return copy;

}

catch (CloneNotSupportedException e) {

return null; // won't ever happen

}

}

}

Copying the list of transactions (and any other modifiable reference fields) produces a deep copy that is independent of the original.

We don’t write the clone method just because we need to override it for the call-by-value or returning from methods or that kind of stuffs. We don’t have those kind of problems in JAVA.

We need it incase somebody makes a copy of our method.

Since they make shallow copying protected, you cannot make copy of your class by accident if you don’t override clone method by implement the Cloneable.

**Effective JAVA Tip**

Override clone judiciously.

Cloning has many gotchas and warts:

– protected vs. public (make clone method public)

– flaws in the presence of inheritance

– requires the use of an ugly tagging interface

– throws an ugly checked exception

– easy to get wrong by making a shallow copy instead of a deep copy

**JAVA EXCEPTIONS**

• Exception – an indication of a problem that occurs during a program’s execution

• Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully

• Exception handling enables programmers to create programs that are more robust and fault-tolerant

**Using the *throws* Clause**

throws clause – specifies the exceptions a method may throw

– Appears after method’s parameter list and before the method’s body

– Contains a comma-separated list of exceptions

– Exceptions can be thrown by statements in method’s body of by methods called in method’s body

– Exceptions can be of types listed in throws clause or subclasses

**CHECK DivideByZeroWithExceptionHandling.java**

**Throwing Exceptions Using the *throw* Statement**

• throw statement – used to throw exceptions

• Programmers can thrown exceptions themselves from a method if something has gone wrong

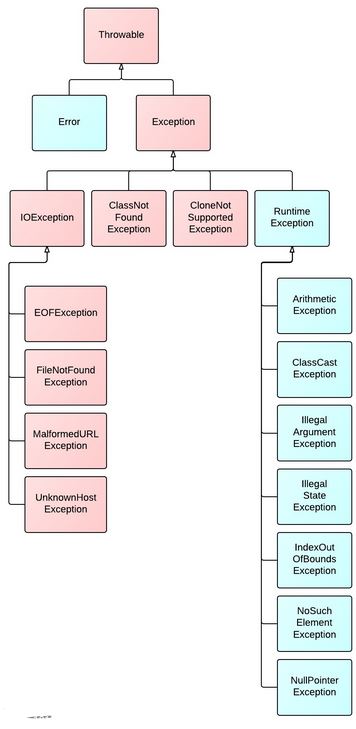
• throw statement consists of keyword throw followed by the exception object

**CHECK UsingExceptions.java**

All the exceptions expands from Exception class.

Whenever you throw an exception of your class, you expands from Exception superclass.

If you are catching anything that expands from Exception then you should be fine.



**GENERICS**

– New feature of J2SE 5.0

– Like templates in C++

– Provide compile-time type safety

• Catch invalid types at compile time

– Generic methods

• A single method declaration

• A set of related methods

– Generic classes

• A single class declaration

• A set of related classes

With a method m1 that takes an Object (m1(Object o)), this method can work on any kind of type. So why you need templates? That was the idea and they didn’t put generics in the beginning of JAVA language.

Later they realize that if people send wrong types to these methods during the runtime, then I have to detect them during the runtime and I have to say that your type is wrong. This would be slow even for JAVA. So generics provide compile-time type safety.

**!!!CHECK GenericMethodTest.java**

Generic class/interface:

public interface List <E>{

void add(E x);

Iterator<E> iterator();

}

public interface Iterator<E> {

E next();

boolean hasNext();

}

All the JAVA collection hierarchy is implemented using generics.