Design Patterns II

Ergude Bao

Beijing Jiaotong University

Contents

- Creational patterns II

Creational Patterns II

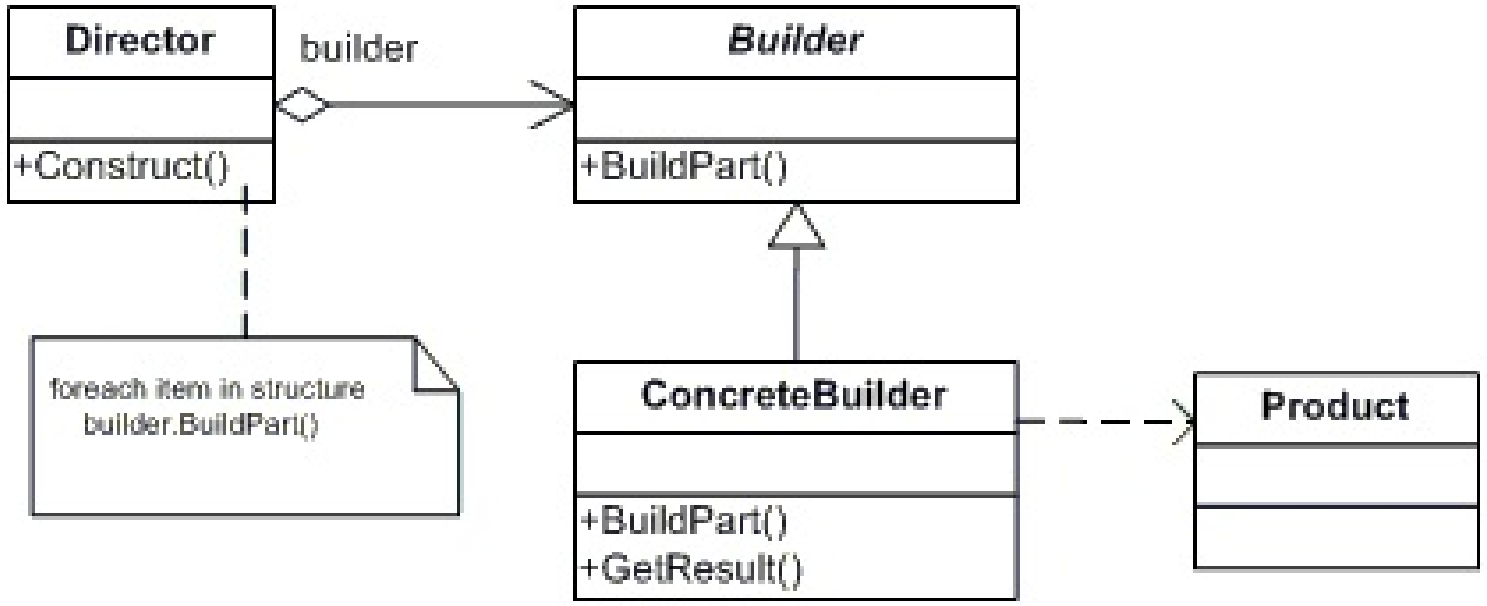
- Builder Pattern
  - Separates the construction of a complex object from its representation so that the same construction process can create different representations
- Prototype Pattern
  - Starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances

Builder Pattern

Builder Pattern

- Motivation
  - We want to assemble objects of several classes into various complex objects
- Solution
  - Separate the common part to construct various complex objects from the specific parts to construct them

Builder Pattern

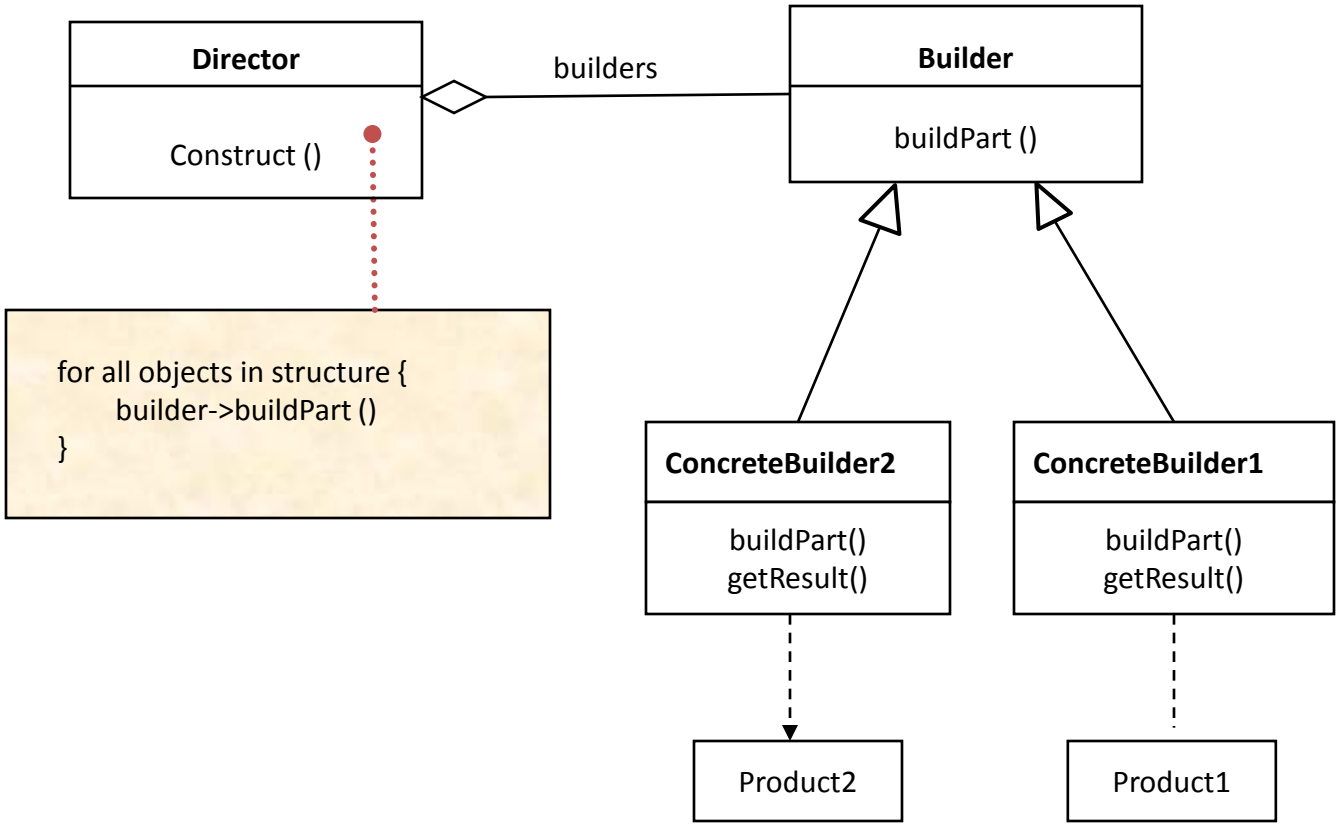


Participants

- Builder
  - Specifies an abstract interface for constructing parts of a Product object
- ConcreteBuilder
  - Constructs parts of the Product object by implementing the Builder interface
  - Provides an interface for retrieving the product
- Director
  - Constructs the Product object using the Builder interface
- Product
  - Represents the complex object under construction

Builder Pattern

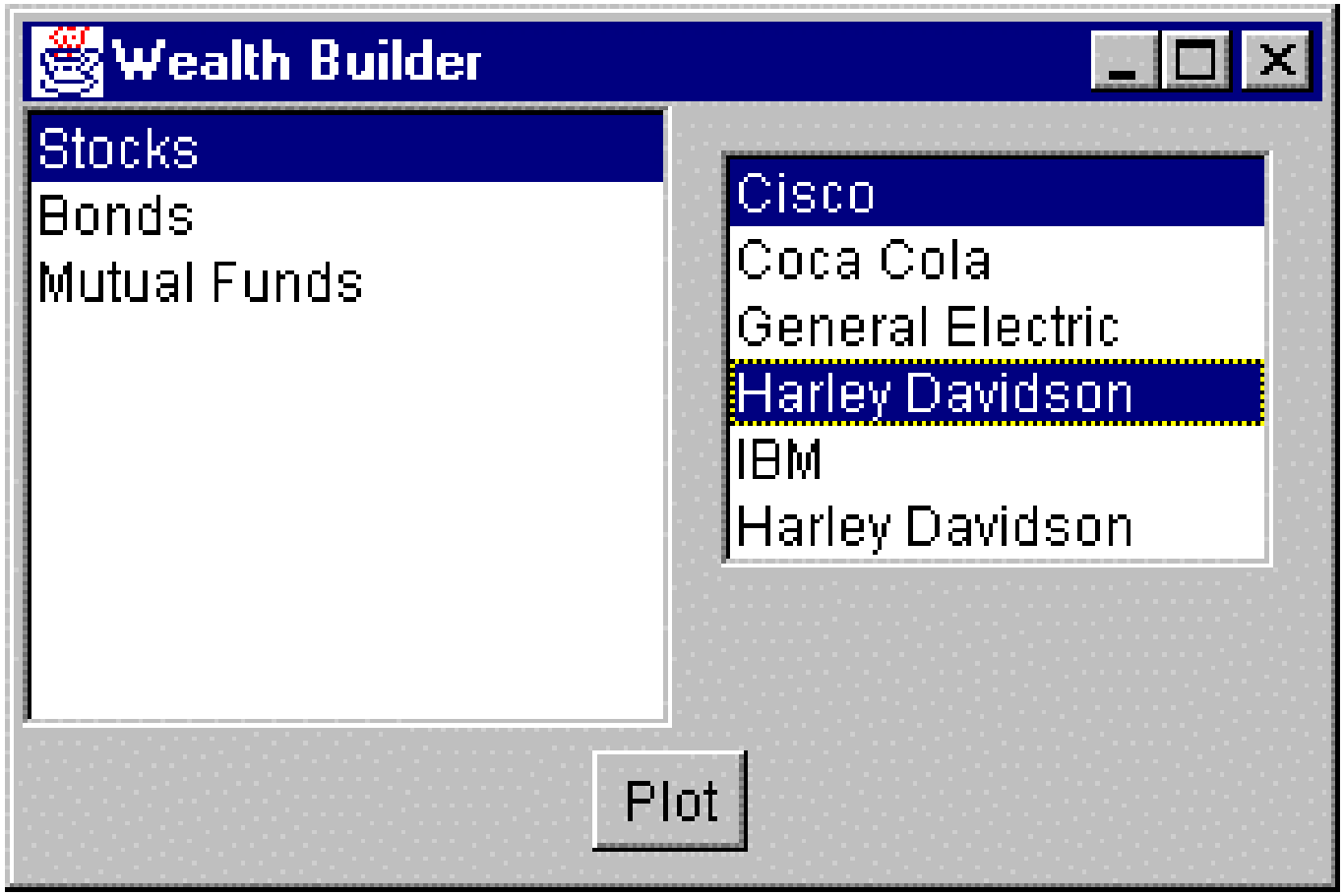
- With two ConcreteBuilders



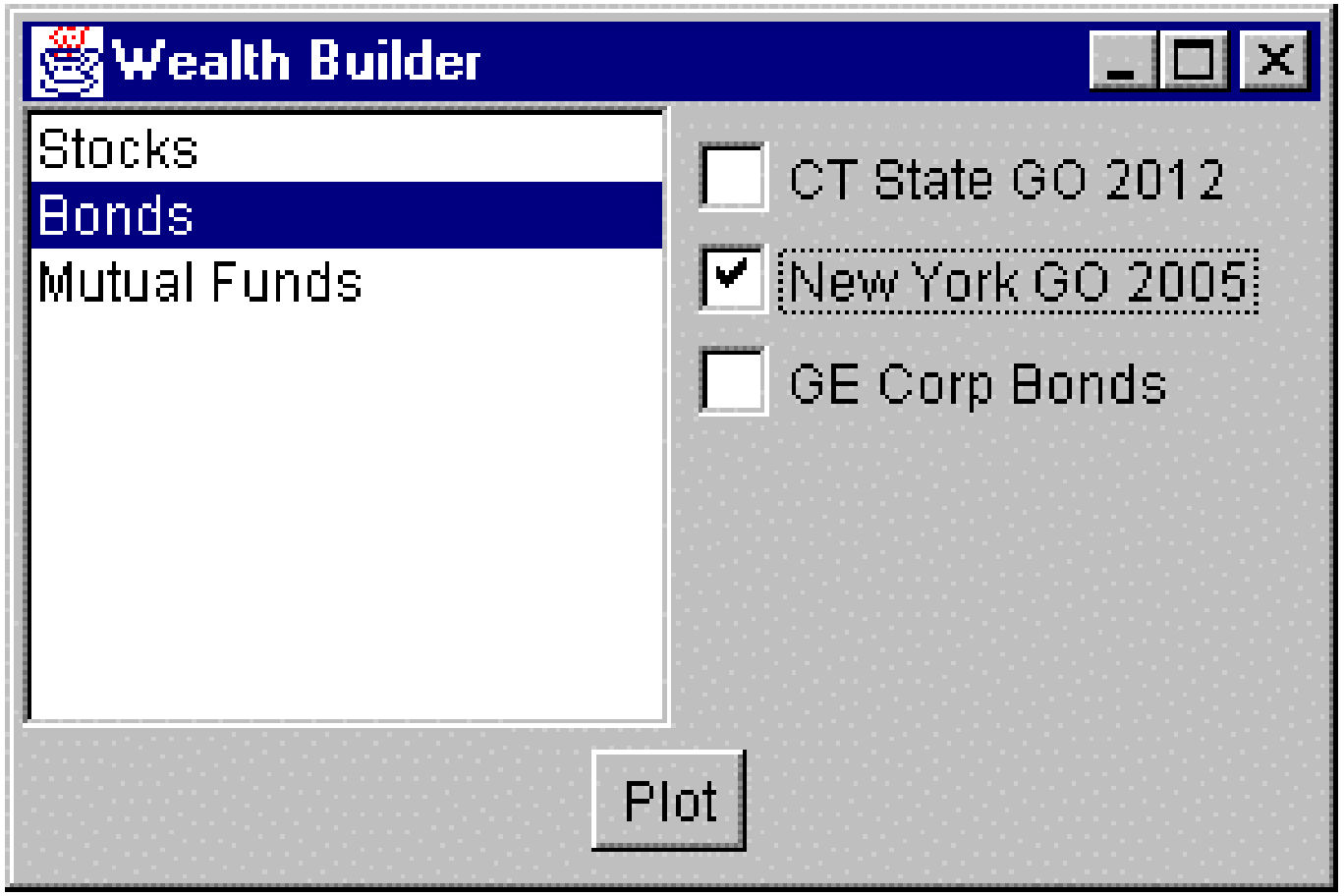
Example

- Write a program to display a list of stocks, bonds and mutual funds in each category so we can select one or more of the investments and plot their comparative performance
  - If there is a large number of funds, use a multi-choice list box
  - If there are 3 or fewer funds, use a set of check boxes

Example


Wealth Builder
Stocks
Bonds
Mutual Funds
Cisco
Coca Cola
General Electric
Harley Davidson
IBM
Harley Davidson
Plot

Example


Wealth Builder
Stocks
Bonds
Mutual Funds
CT State GO 2012
New York GO 2005
GE Corp Bonds
Plot

Example

```
abstract class multiChoice {
        //this is the abstract base class that
        //the listbox and checkbox choice  panels are derived from
        Vector choices; //array of labels

        public multiChoice(Vector choiceList) {
                        choices = choiceList; //save list
        }
        //to be implemented in derived classes

        abstract public Panel getUI();
        //return a Panel of components

        abstract public String[] getSelected();
        //get list of items

        abstract public void clearAll();
        //clear selections
}
```

Example

```
class listboxChoice extends multiChoice {
        List list; //investment list goes here

        public listboxChoice(Vector choices)  {super(choices);}

        public Panel getUI() {
        //create a panel containing a list box
                Panel p = new Panel();
                list = new List(choices.size()); //list box
                list.setMultipleMode(true); //multiple
                p.add(list);
                //add investments into list box
                for (int i=0; i< choices.size(); i++)
                        list.addItem((String)choices.elementAt(i));
                return p; //return the panel
        }

        public String[] getSelected() {
                int count =0;
                //count the selected listbox lines
                for (int i=0; i < list.getItemCount(); i++ ) {
                        if (list.isIndexSelected(i)) count++;
                }
                String[] slist = new String[count];
                //copy list elements into string array
                int j = 0;
                for (int i=0; i < list.getItemCount(); i++ ) {
                        if (list.isIndexSelected(i)) slist[j++] = list.getItem(i);
                }
                return(slist);
        }
}
```

Example

```
…
public checkBoxChoice(Vector choices) {
        super(choices);
        count = 0;
        p = new Panel();
}

public Panel getUI() {
        String s;
        //create a grid layout 1 column by n rows
        p.setLayout(new GridLayout(choices.size(), 1));
        //and add labeled check boxes to it
        for (int i=0; i< choices.size(); i++) {
                s =(String)choices.elementAt(i);
                p.add(new Checkbox(s));
                count++;
        }
        return p;
}
…
```

Example

- Create a simple factory class that decides which of these two classes to return

```
class choiceFactory {
        multiChoice ui;
        public multiChoice getChoiceUI(Vector choices){
                if(choices.size() <=3)
                        ui = new checkBoxChoice(choices);
                else
                        ui = new listboxChoice(choices);
                return ui;
        }
}
```

Example

```
public wealthBuilder() {
    super("Wealth Builder"); //frame title bar
    setGUI(); //set up display
    buildStockLists(); //create stock lists
    choiceFactory cfact; //the factory
}
```

Example

```
private void setGUI() {
        setLayout(new BorderLayout());
        Panel p = new Panel();
        add("Center", p);
        //center contains left and right panels
        p.setLayout(new GridLayout(1,2));
        //left is list of stocks
        stockList= new List(10);
        stockList.addItemListener(this);
        p.add(stockList);
        stockList.add("Stocks");
        stockList.add("Bonds");
        stockList.add("Mutual Funds");
        stockList.addItemListener(this);
        //Plot button along bottom of display
        Panel p1 = new Panel();
        p1.setBackground(Color.lightGray);
        add("South", p1);
        Plot = new Button("Plot");
        Plot.setEnabled(false);
        //disabled until stock picked
        Plot.addActionListener(this);
        p1.add(Plot);
        //right is empty at first
        choicePanel = new Panel();
        choicePanel.setBackground(Color.lightGray);
        p.add(choicePanel);
}
```

Example

```
private void stockList_Click() {
        Vector v = null;
        int index = stockList.getSelectedIndex();
        choicePanel.removeAll();
        //remove previous ui panel
        //this just switches among 3 different Vectors and passes the one you select to the Builder
        switch(index) {
                case 0:
                        v = Stocks; break;
                case 1:
                        v = Bonds; break;
                case 2:
                        v = Mutuals;
        }
        mchoice = cfact.getChoiceUI(v);
        choicePanel.add(mchoice.getUI());
        choicePanel.validate();
        Plot.setEnabled(true);
}
```

When to Use

- One construction algorithm and many representations for a complex object
- Algorithm and representations are independent

Advantages

- Allows varying the internal representation of the complex object to build
- Hides the details of how a part of the object is built
- Improves modifiability and maintainability, since each builder is independent from the others and the rest of the program

Builder Pattern vs. Abstract Factory Pattern

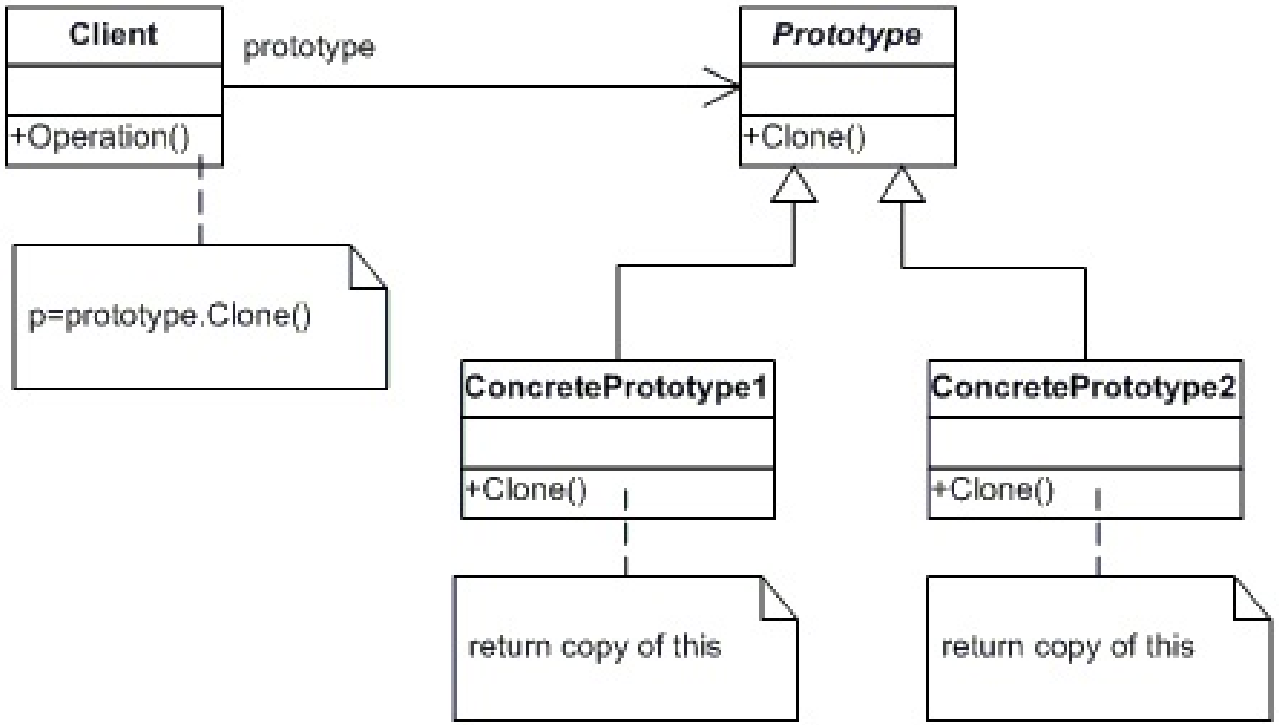
- Similarity: Both create a set of objects
- Difference: abstract factory pattern creates objects of a family of related classes, while builder pattern creates objects to construct a complex object

Prototype Pattern

Prototype Pattern

- Motivation
  - Creating an instance of a class can be very time-consuming or complex
- Solution
  - Make copies of the original instance and modify them as appropriate

Prototype Pattern



Participants

- Prototype
  - Declares an interface for cloning itself
- ConcretePrototype
  - Implements an operation for cloning itself
- Client
  - Creates a new object by asking a prototype to clone itself

Shallow Copy vs. Deep Copy

- Similarity: both build a new instance
- Difference: shadow copy only copies the reference to an instance inside the original instance, while deep copy copies both the reference and the instance inside the original instance

Shallow Copy

- Make a shallow copy of any Java object using the clone method

```
Jobj j1 = (Jobj)j0.clone();
```

- Four restrictions on the clone method
  - The clone method always returns an object of type Object. You must cast it to the actual type of the object you are cloning
  - It is a protected method and can only be called from within the same class or the module that contains that class
  - You can only clone objects which are declared to implement the Cloneable interface
  - Objects that cannot be cloned throw the CloneNotSupported Exception

Shallow Copy

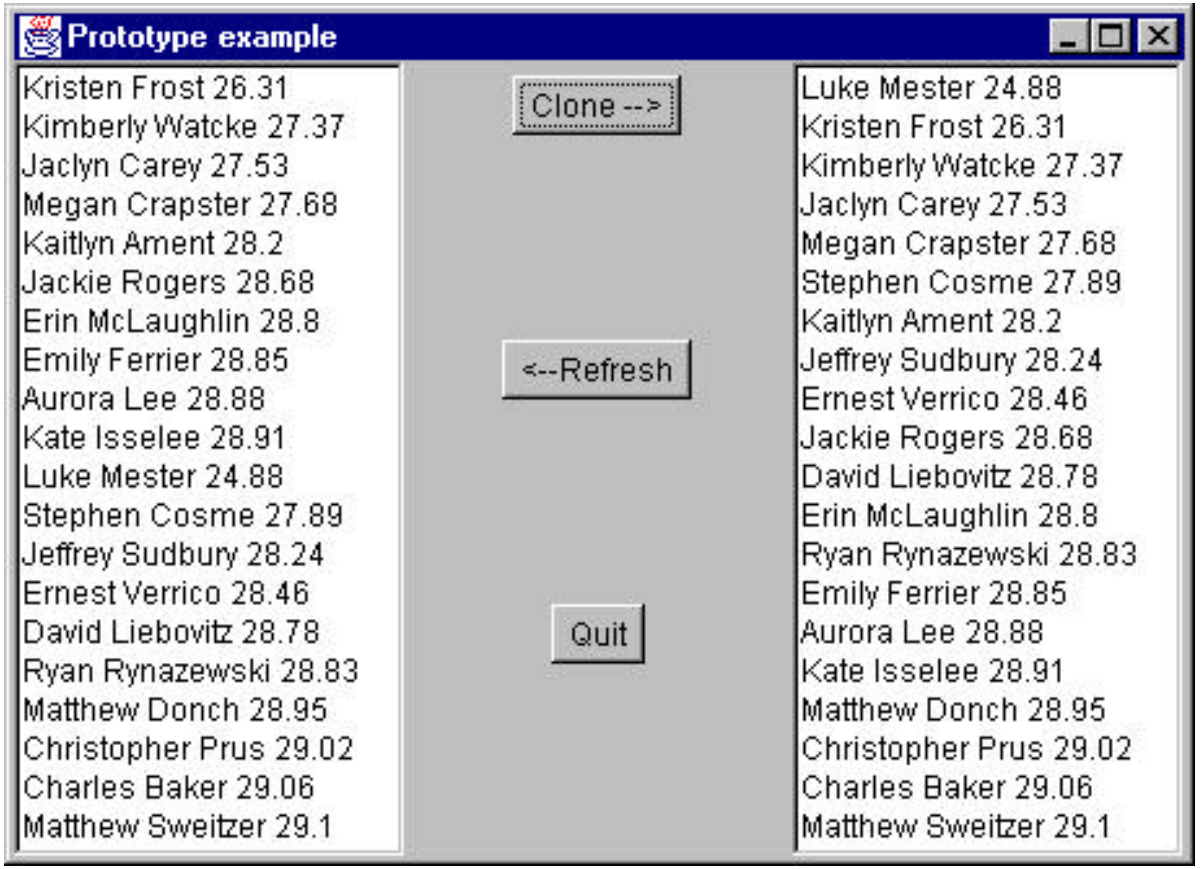
```
public class SwimData implements Cloneable {
	public Object clone() {
		try{
			return super.clone();
		} catch(Exception e) {
			System.out.println(e.getMessage());
			return null;
		}
	}
}
```

Shallow Copy

- It is possible to do the typecasting within the method replacing clone() for cloneMe()

```
public SwimData cloneMe() {
    try{
        return (SwimData)super.clone();
    } catch(Exception e) {
        System.out.println(e.getMessage());
        return null;
    }
}
```

Example


Prototype example
Kristen Frost 26.31
Kimberly Watcke 27.37
Jaclyn Carey 27.53
Megan Crapster 27.68
Kaitlyn Ament 28.2
Jackie Rogers 28.68
Erin McLaughlin 28.8
Emily Ferrier 28.85
Aurora Lee 28.88
Kate Isselee 28.91
Luke Mester 24.88
Stephen Cosme 27.89
Jeffrey Sudbury 28.24
Ernest Verrico 28.46
David Liebovitz 28.78
Ryan Rynazewski 28.83
Matthew Donch 28.95
Christopher Prus 29.02
Charles Baker 29.06
Matthew Sweitzer 29.1
Clone -->
<--Refresh
Quit
Luke Mester 24.88
Kristen Frost 26.31
Kimberly Watcke 27.37
Jaclyn Carey 27.53
Megan Crapster 27.68
Stephen Cosme 27.89
Kaitlyn Ament 28.2
Jeffrey Sudbury 28.24
Ernest Verrico 28.46
Jackie Rogers 28.68
David Liebovitz 28.78
Erin McLaughlin 28.8
Ryan Rynazewski 28.83
Emily Ferrier 28.85
Aurora Lee 28.88
Kate Isselee 28.91
Matthew Donch 28.95
Christopher Prus 29.02
Charles Baker 29.06
Matthew Sweitzer 29.1

Example

- Left-hand list box
  - Loaded when the program starts
  - Display the original data
  - Names are sorted by gender and then by time
- Right-hand list box
  - Loaded when you click on the Clone button
  - Display the sorted data in the cloned class
  - Names are sorted only by time

Example

```
class Swimmer {
    String name;
    int age;
    String club;
    float time;
    boolean female;
…
```

Example

```
public class SwimData implements Cloneable {
        Vector swimmers;

        public SwimData(String filename) {
                String s = "";
                swimmers = new Vector();
                //open data file
                InputFile f = new InputFile(filename);
                s= f.readLine();
                //read in and parse each line
                while(s != null) {
                        swimmers.addElement(new Swimmer(s));
                        s= f.readLine();
                }
                f.close();
        }
…
                swList.removeAll(); //clear list
                for (int i = 0; i < sdata.size(); i++) {
                        sw = sdata.getSwimmer(i);
                        swList.addItem(sw.getName()+" "+sw.getTime());
                }
…
                sxdata = (SwimData)sdata.clone();
                sxdata.sortByTime(); //re-sort
                cloneList.removeAll(); //clear list
                //now display sorted values from clone
                for(int i=0; i< sxdata.size(); i++) {
                        sw = sxdata.getSwimmer(i);
                        cloneList.addItem(sw.getName()+"  "+sw.getTime());
                }
…
```

Deep Copy

- Make a deep copy using the serializable interface
  - A seriablizable class can be written out as a stream of bytes and those bytes can be read back to reconstruct the class
- Deep Copy allows to copy and get a completely independent instance from the original of any complexity

Deep Copy

```
public Object deepClone() {
        try{
                ByteArrayOutputStream b = new ByteArrayOutputStream();
                ObjectOutputStream out = new ObjectOutputStream(b);
                out.writeObject(this);
                ByteArrayInputStream bIn = new ByteArrayInputStream(b.toByteArray());
                ObjectInputStream oi = new ObjectInputStream(bIn);
                return (oi.readObject());
        } catch (Exception e) {
                System.out.println("exception:"+e.getMessage());
                return null;
        }
}
```

When to Use

- The system is independent from how its products are created, composed, and represented, so that classes can be instantiated are specified at run-time
- It is more convenient to copy an existing instance than to create a new one

Summary of Creational Patterns

- Factory Pattern is used to choose and return an instance of a class from a number of similar classes based on data you provide to the factory
- Abstract Factory Pattern is used to return one of several groups of classes. In some cases it actually returns a Factory for that group of classes
- Singleton Pattern is a pattern that insures there is one and only one instance of an object, and that it is possible to obtain global access to that one instance

Summary of Creational Patterns

- Builder Pattern assembles a number of objects to make a new object, based on the data with which it is presented. Frequently, the choice of which way the objects are assembled is achieved using a Factory
- Prototype Pattern copies or clones an existing class rather than creating a new instance when creating new instances is more expensive