

© nabg

# OO usage

- © nabg
- ## OO usage
- **An example:**
    - Limited (several classes implementing an interface, no real hierarchy)
    - A toy (just a game)
    - Exists so that you can see some actual code using heterogeneous collections etc and can visualize different behaviours for different classes
  - **Discussion of where you might use OO**
    - Application specific hierarchy – does occur, but not that common
    - Application specific subclasses for use within an existing application framework – most likely place where you will use OO

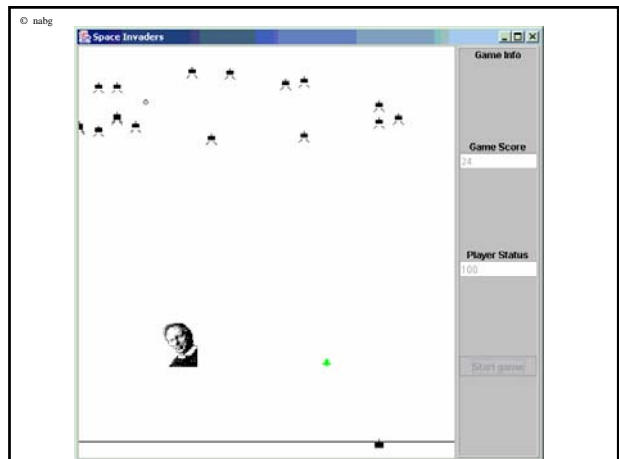
© nabg

## OO Toy Example

© nabg

# SpaceInvaders

- © nabg
- ## SpaceInvaders ? ! !!
- You must have played some version of this game at sometime in your life.
  - Some of you will even have a Java version in your mobile phones.



## Invaders

- Invaders come in many different forms
- Some commonalities of behaviours, some differences
- From perspective of game code framework (and of player) they are all the same – just pesky invaders that must be destroyed

## Game-1

- Details do vary but overall approach common
  - You have a "gun"
    - Appears at the bottom of the screen,
    - Can be moved horizontally with some control
    - Can fire (mouse click)
    - Fire destroys invaders located above gun; score points for each invader destroyed
    - Gun can suffer incremental damage by invaders
    - When gun destroyed, may get a replacement gun, or maybe end of game

## Game-2

- Details do vary but overall approach common
  - Invaders
    - Most appear at the top of the screen and work their way down, though some have different flight patterns (e.g. moving from side to side across the screen).
    - The typical invader descends until "ground level" (gun level) when it explodes damaging the gun.

## Game-3

- Objective of game
  - Destroy as many invaders as possible and so achieve highest score before all guns destroyed.

## Specifics of this version

- Two "threads" in program
  - One handles graphics aspects
  - Second handles game
    - Handle gun actions
    - For each invader handle invader actions
- Game keeps constant number of invaders – if an invader is destroyed in one cycle of loop it is replaced at start of next cycle
- Start with all invaders of same type, new invaders are of differing types chosen randomly

## Illustrates

- Invader class hierarchy
  - All invaders do the same thing
    - Move
    - Seek to destroy player
  - Each class of invader has its own approach
- Heterogeneous collections
- Dynamic binding
- Polymorphic object reference
- Invader "factory"
- ...

© nabg

## Limited example

- Very light weight example
  - No real class hierarchy
  - No partially implemented abstract classes
- Invader interface
- Some concrete invader classes

© nabg

## Code

- Driver program
- Graphic components
- Threads
  - One looks used by AWT graphics system
  - Other one runs game

---

- Invaders
  - Classes
  - Invader factory

*This code written entirely in terms of use of "SpaceInvader" abstraction*

*This part of code deals with different kinds of Invaders*

© nabg

## Use of SpaceInvader abstraction

- Illustrates important aspect of OO programs
  - Knowledge of specialized subclasses limited to very small well defined part of overall code
  - Most code works simply with the abstract class
- Consequence:
  - original specification is not hard coded into code in form of "switch ... case ... case ..." statements
  - program easily extended to work with subsequently defined subclasses

© nabg

### Main driver loop

```

while(gun's status > 0){
  checkInvaders - replace any invaders that were destroyed in last cycle
  adjustGunLocation - based on mouse motion events recorded separately
  fireGun - based on occurrence of mouse click recorded separately; this operation may damage or destroy an invader located above the gun (the first invader hit takes all damage and shields other invaders);
  ...
}
  
```

© nabg

### Main driver loop

```

while(gun's status > 0){
  ...
  for each invader in collection
    invader move
    check if invader is still "alive" –
      if not, remove it from the collection
    check whether gun destroyed by this invader, terminate main loop as appropriate
  get control panel to update score and status values
  request a repaint of the game panel
  delay a little before next cycle through this loop
}
  
```

© nabg

## Heterogeneous collection ...

- Game uses a Vector ("myInvaders) to store invaders

```

Iterator iter = myInvaders.iterator();
while(iter.hasNext()) {
  SpaceInvader si = (SpaceInvader) iter.next();
  si.move();
  if(!si.live())
    iter.remove();// need iterator rather than for-each loop
  if(myStatus<=0) {
    myGameInProgress = false; break; }
}
  
```

## Polymorphic reference Dynamic dispatch

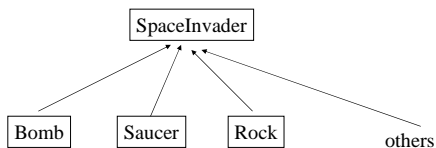
```
SpaceInvader si =
    (SpaceInvader) iter.next();
si.move();
```

- si – the polymorphic reference
  - References SpaceInvader objects of different forms at different times
- si.move() – dynamic binding
  - Find the move function associated with the current SpaceInvader object


## SpaceInvader

```
interface SpaceInvader {
    void move();
    void paint(java.awt.Graphics g);
    boolean live();
    boolean isHit(int location);
    int value();
}
```

## Invader classes



## SpaceInvaders

- Every SpaceInvader can
  - Move
    - Bombs move straight down accelerating as they go
    - Rocks tumble slowly down
    - Saucers jig-left, jig-right, descend slowly
  - Paint
    - Bombs paint themselves as leading wedges, rocks as little circles, saucers as 
  - Live
  - isHit, value – functions that really differ only in parametric values

## InvaderFactory

- Really the only part of the code that needs to know about different invader classes, *(and there Java tricks that makes it possible for even this part of the code to avoid hard-coding of the set of possible Invader choices)*
- Randomly picks an Invader subtype
- Creates instance of that type
- Completes any other initializations

```
public class InvaderFactory {
    private static Random sRandgen = new Random();
    public static SpaceInvader defaultInvader(Game g)
    {
        return new Invader1(g);
    }
    public static SpaceInvader newInvader(Game g){
        SpaceInvader si = null;
        int choice = sRandgen.nextInt(20);
        if(choice==0)si = new Invader1(g);
        else
        if(choice<4) si=new Invader2(g);
        else
        if(choice<10) si = new Invader3(g);
        else
        if(choice<15)si= new Invader4(g);
        else si = new Invader5(g);
        return si;
    }
}
```

## Knowing about subclasses

- Factory, as just shown, references Invader subclasses (Invader1, Invader2, ...) explicitly
  - OK, better here at single point rather than all over program
  - But will need to edit and recompile this class if do add another invader subclass.
- Factory style as shown is quite common for C++, Java etc

## Java feature

- In Java, one can avoid entirely the incorporation details of classes into program instead such information becomes *data*.
- At start-up, program reads a text data-file with the names of the classes
- There is a Java construct to create an instance of class knowing just class name
- In this example, InvaderFactory class would read a file with Invader class names and relative frequencies. Its newInvader method would pick class based on these frequencies

## java.lang.Class.forName(...)

- java.lang.Class
  - Instance of class Class exists for every class that you use in your program
  - It is created (by Class Loader) when you first make use of a class
    - Normally this means when at execution time you have code that creates an instance of that class or when you invoke a static method of the class.
  - Contains “virtual table” (pointers to function code) and information on methods
- Has static method classForName(String className)
  - If named class not previously encountered, uses class loader to load the code and create the class Class object
  - Returns reference to class Class object

## newInstance()

- Class Class (!) has instance member “newInstance”
  - Creates a new instance of that class (a default no-argument constructor must exist)

## InvaderFactory using Java features

- On initialization, read data file with names and relative frequencies of different invader subclasses (invaderNames[])
  - In newInvader
    - Randomly pick an invader name (using frequencies to guide random choice)
- ```
int which = some code to pick invader type;
String invaderClassName = invaderNames[which];
Class invaderClass =
    Class.forName(invaderClassName);
Object invaderObject = invaderClass.newInstance();
SpaceInvader si = (SpaceInvader) invaderObject;
si.initialize(g); /* no argument constructor, so must
                  explicitly initialize */
```

## Why not general? Why not C++?

- This technique requires dynamic linking of code
  - Possible (on most platforms) in C++ but difficult
  - Non-standard, need lots of platform specific code conditionally compiled into program

## Invader classes

- Implement SpaceInvader interface
  - All functions declared in interface get effective implementations, these are distinct in each Invader class (though may be similar).
  - (as noted earlier, bit limited example; usually, there would be some intermediate partially implemented abstract class in the hierarchy)

## Example: bomb class

- Starts at some random x-location “high” above ground
- Moves down, accelerating
- Gets destroyed by gun’s fire if approximately above gun (bomb’s centre within  $\pm 5$  units from gun’s centre)

## Defining your Invader in NetBeans : 1

```
public class SimpleBomb {
    /** Created a new instance of SimpleBomb */
    public SimpleBomb() {
    }
}

+ Source code
(spaceInvaders.SimpleBomb is not abstract and does not override abstract method value() in spaceInvaders.SpaceInvader)
public class SimpleBomb implements SpaceInvader {
    /** Created a new instance of SimpleBomb */
    public SimpleBomb() {
    }
}
```

## Defining your Invader in NetBeans : 2

```
public class SimpleBomb implements SpaceInvader {
    /** Created a new instance of SimpleBomb */
    public SimpleBomb() {
    }
    public void move() {
    }
    public void paint(Graphics g) {
    }
    public boolean live() {
    }
    public boolean isHit(int location) {
    }
    public int value() {
    }
}
```

```
public class SimpleBomb implements SpaceInvader
{
    private int x;
    private int y;
    private int yvelocity;
    private int state;
    private Random rand = new Random();
    private Game theGame;
    public SimpleBomb(Game aGame){
        theGame = aGame;
        x = ...; /* some random x location -100...+100 */
        y = 500 + rand.nextInt(100);
        yvel = -1; /* Going down, velocity negative */
        state = 1;
    }
    ...
}
```

```
public class SimpleBomb implements SpaceInvader {
    ...
    public void move() {
        y += yvel; /
        yvel--;
        if (y < 0) theGame.groundBurst(15);
    }
    public void paint(Graphics g) {
        /* Convert game coords to graphic coords */
        int xg = theGame.X2X(x);
        int yg = theGame.Y2Y(y);

        g.fillRect(xg-4, yg-2, 9, 5);
        g.drawLine(xg, yg-2, xg, yg-5);
        g.drawLine(xg-3, yg+5, xg-6, yg+8);
        g.drawLine(xg+3, yg+5, xg+6, yg+8);
    }
    ...
}
```

```

© nabg

public class SimpleBomb implements SpaceInvader {
    ...
    public boolean live()
    { return (state>0) && (y>0); }
    public int x() { return x; }

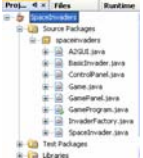
    public boolean isHit(int location) {
        boolean hit = false;
        int diff = Math.abs(location - x());
        if(diff<5){ hit = true; state=0; }
        return hit;
    }
    public int value() { return 25; }
}

```

## Rest of SpaceInvaders application

- Side issue, not part of OO introduction
- Just illustrate some fragments of the code so that you get some idea of how it works
- Uses aspects of Java not yet covered
  - Graphics
  - Threads

## Invader classes



## SpaceInvaders : the classes

- A2
  - Main driver program
- A2GUI
  - User interface
    - A region where game activities displayed (GamePanel)
    - A region with controls and status display fields (ControlPanel)
    - Regions fitted into interface
- GamePanel & ControlPanel
  - User interface classes
- Game
  - Game mechanics, based purely on SpaceInvader abstraction
- InvaderFactory, and Invaders hierarchy

## SpaceInvaders: A2 class

- A2:
  - main()
    - Create Game object
    - Create graphic interface
      - A JFrame (kind of window) that holds an A2GUI panel
  - Couple of utility functions for loading sounds and images

```

© nabg
import ...; /* Lots of imports - javax.swing etc */
public class A2 {
    private static java.awt.Toolkit toolkit;
    private static Component displayer;
    public static Image getImage(String name) {
        ...
        return anImage;
    }
    public static AudioClip getSound(String name) {
        ...
        return theClip;
    }
    ...
}
Exist to let Invaders use sound effects and images

```

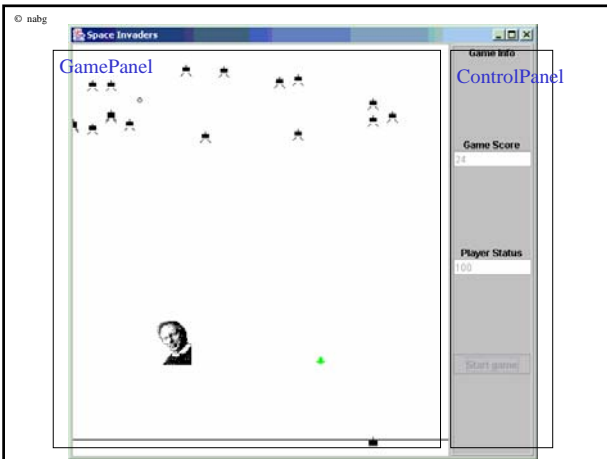
```

© nabg
public class A2 {
    ...
    public static void main(String[] args) {
        try {
            ... Window management code
            Game aGame = new Game(20);
            JFrame aFrame = new JFrame("Space Invaders");
            aFrame.addWindowListener(1);
            A2GUI displayComponent = new A2GUI(aGame);
            ...
            aFrame.getContentPane().add(displayComponent);
            aFrame.setSize(new Dimension(640,640));
            aFrame.show();
        } catch(Exception e) { ... }
    }
}

```

## A2GUI class

- Creates the Control and Game panel objects, then arranges them side by side
  - Arrangement of components in a GUI is done with a “layout manager”, in this case a GridBagLayout
  - GridBagLayout – allows design of complex interfaces, but coding is very laborious (not hard, just a matter of defining lots of constraints that specify precisely how GUI components are placed)



```

© nabg
public class A2GUI extends JPanel {
    private GamePanel gp;
    private ControlPanel cp;
    private Game myGame;
    public A2GUI(Game aGame){
        myGame = aGame;
        gp = new GamePanel(aGame);
        cp = new ControlPanel(aGame);
        cp.setBorder(BorderFactory.createEtchedBorder(
            EtchedBorder.RAISED));
        GridBagLayout gbl = new GridBagLayout();
        GridBagConstraints gbc =
            new GridBagConstraints();
        setLayout(gbl);
        gbc.weightx = 100;
        ...
        add(gp, gbc); Define constraints for placing GamePanel
        ...
        add(cp, gbc); Then the same for ControlPanel
    }
}

```

## ControlPanel class

- Builds a display with text output fields and an action button spaced out in a vertical column
  - Again, GridBagLayout used to control layout
- Has public functions that allow update of score and status fields.
- Control button –
  - Initially enabled
  - Use: “starts” game (this actually starts another thread), and disables button
  - End of game (thread terminates): re-enable game button

```

© nabg
class ControlPanel extends JPanel implements ActionListener {
    private Game myGame;
    private JButton playButton;
    private JTextField score;
    private JTextField status;
    public ControlPanel(Game aGame) {
        builds the GUI interface - quite long winded
        code to place elements where desired
    }
    public void actionPerformed(ActionEvent aev) {
        playButton.setEnabled(false);
        myGame.start();
    }
    public void update()
    {
        score.setText(Integer.toString(myGame.score()));
        status.setText(Integer.toString(myGame.status()));
    }
    public void reEnableGame() { playButton.setEnabled(true); }
}

```

© nabg

```

public ControlPanel(Game aGame) {
    myGame = aGame;
    myGame.setControlLink(this);
    GridBagLayout gbl = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();
    setLayout(gbl);
    gbc.weightx = 0;
    ...
    add(new JLabel("Game Info"), gbc);
    ...
    playButton = new JButton("Start game");
    add(playButton, gbc);
    playButton.addActionListener(this);
    ...
}

```

*Little of interest in graphics set up code!*

*One thing to note – as is very common – display (view) objects like these are linked to data (model) objects like the game.*

*Game has link to control panel – so it can update fields.*

*Control panel has link to game – so it can start thread.*

© nabg

## GamePanel

- Provides an area where game can “paint” its picture.
- Arranges that Game object
  - listens to mouse-motion events associated with this display area (these control position of gun)
  - Listens for mouse-button events (represent firing of gun)

© nabg

```

class GamePanel extends JPanel {
    public Game myGame;
    public void setLinkoGame(Game aGame)
    { myGame = aGame; }
    public void paint(Graphics g){ myGame.paint(g); }
    public GamePanel(Game aGame){
        myGame = aGame;
        myGame.setGamePanellink(this);
        this.addMouseListener(myGame);
        this.addMouseMotionListener(myGame);
    }
}

```

© nabg

## Game

- **Owns**
  - Collection of invaders, data defining gun
  - Links to display elements
- **Does**
  - Manage coordinate systems
  - Paint image of game with invaders and gun
  - Create new thread for game when required to by control panel
  - Run the game
  - Accept mouse events (if running game)
    - Adjust gun position
    - Fire

© nabg

## Coordinates (minor detail)

- Window coordinates:
  - (0,0) is top-left corner
  - X-increase across screen
  - Y-increase DOWN screen
- Application coordinate system often different, e.g. here
  - “0” level is ground-level where gun is
  - Top of window is 500-metre level where sight invaders
  - X=0 is middle of screen
- One has to map between coord systems, here done by helper functions in Game

© nabg

```

public class Game
    implements Runnable, MouseListener,
        MouseMotionListener {
    ...
    private Vector      myInvaders;
    private int         myGunLocation;
    ...
    private AudioClip  myGameOverSound;
    public int X2X(int gameX){ ... }
    public int Y2Y(int gameY) { ... }
    public Game(int num){ ... }
    public void setControlLink(ControlPanel cp){ ... }
    public void setGamePanellink(GamePanel gp){ ... }
    public int score() { return myScore; }
    public int status() { return myStatus; }
    public int gunLocation() { return myGunLocation; }
    public GamePanel getGamePanel() { ... }
    public void paint(Graphics g) { ... }
    public void start() { ... }
    ...
}

```

```

© nabg
public class Game
...
private void checkInvaders() { ... }
public void zapGun() { myStatus -= 1; }
public void groundBurst(int damage) { ... }
public void run() { ... }
public void mouseClicked(MouseEvent e) { ... }
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}

public void mouseDragged(MouseEvent e){}
public void mouseMoved(MouseEvent e){ ... }
private void adjustGunLocation(){ ... }
private void fireGun() { ... }
}

```

```

© nabg
public void paint(Graphics g) {
...
// Paint each invader
for(SpaceInvader si : myInvaders)
    si.paint(g);
}

```

Typical OO style – for each SpaceInvader, go draw yourself as you want!  
myInvaders – heterogeneous collection  
si – polymorphic pointer  
si.paint() – dynamic dispatch

super

Getting your parent (class) to do some work for you

### Feature missing from SpaceInvaders

- *Invoking “super” methods*
- This is common feature when have class hierarchy
  - Pure abstract class (Java interface)
  - Partially implemented abstract class
    - Default methods
    - **Partial methods**
  - Concrete classes

### Invent extended Invader hierarchy to illustrate

- interface SpaceInvader
  - Main functions, add “read” and “write” functions
- abstract class BasicInvader
  - Define some common data elements (all invaders have these)
    - Value, Status, (x,y) coords
  - Define any final and any default functions

```

final int value() { return _value; }
boolean live() { return _status > 0; }

```
  - Define “partially implemented functions”
- concrete subclasses Rock, Bomb, Saucer, ...

### Final, default, partial

- Final
  - Only possible implementation fixed

```

final int value() { return _value; }

```
- Default

```

boolean live() { return _status>0; }

```

  - Suppose there is a subclass that has a finite lifetime (e.g. Saucers disappear after being present for 15 rounds – they have an extra lifecount data member)

```

boolean live() { _lifecount++;
return ((_lifecount<15) && (_status>0)); }

```
  - New function defined in Saucer class completely replaces default function defined by BasicInvader
- Partial
  - Abstract class defines some common code used by all subclasses

## “Partially implemented functions”

- Can define part of code at level of abstract class, but subclasses need additional code.
- E.g. Suppose want to read invader from file or write invader to file.
- Data in (text) file
  - Class-name
  - Value, status
  - X, Y
  - Other data specific to subclass (e.g. bombs might have different blasting power, saucers have a lifetime and maybe a color)

## Code to read in invaders

```
BufferedReader input = ...;
Vector myInvaders = ...;

for(;;) {
    String className = input.readLine();
    Class invaderClass = Class.forName(className);
    SpaceInvader si =
        (SpaceInvader) invaderClass.newInstance();
    si.read(input);
    myInvaders.add(si);
    ...
}
```

## Code to write Invaders

```
PrintWriter output = ...;

Iterator iter = myInvaders.iterator();
while(iter.hasNext()) {
    SpaceInvader si =
        (SpaceInvader) iter.next();
    si.write(output);
    ...
}
```

## What would an invader have to write?

- Its class
 

```
output.println(this.getClass().getName());
```
- Its value, and its status
 

```
output.println(_value);
output.println(_status);
```
- Its coordinates etc
 

```
output.println(...)
```
- Its own special data
 

```
output.println(_lifetime);
```
- Common code can go in abstract class and be invoked from subclass

## Example code

```
abstract class BasicInvader {
    protected int _value;
    ...

    void write(
        PrintWriter output)
    {
        out.println(_value);
        ...
    }
}

public class Saucer extends BasicInvader
{
    private int _lifetime;
    ...

    void write(
        PrintWriter output)
    {
        output.println(
            this.getClass().getName());
        super.write(output);
        out.println(_lifetime);
        ...
    }
}
```

## Invoking “super” method

- Function defined in concrete class will invoke method defined in base class at point where it needs the standard processing performed.
- Output
  - Write own class
  - Invoke `super.write` (to save common data)
  - Write special data
- Input
  - Invoke `super.read`
  - Read own special data.

## Re-iterating : interface

- **Interface**
  - Declares a set of function prototypes (no implementations);
    - All classes that implement interface must provide implementations of these functions!
  - Sometimes (rarely) defines constants
  - Role is to describe a “type”, a set of operations, at the most general level

## Re-iterating : abstract class

- Abstract class / partially implemented abstract class / base class
  - Declares some abstract methods that must be defined in concrete subclasses
  - Defines some methods
    - Default implementation
    - Partial implementation
    - Final implementation

## Re-iterating : default implementation

- Abstract class provides a definition for a method
  - Concrete sub-classes will use this definition by default
  - Any concrete sub-class can provide an alternative definition if the default behaviour is inappropriate

## Re-iterating : Partial implementation

- Abstract base class can define partial implementation
  - Define the operations that will be common to all subclasses, e.g.
    - Read/write those data members defined in the abstract class
  - Subclasses define their own versions of same operations
    - Subclass code invokes base class implementation via `super`

## Re-iterating : final implementation

- Author of base class may wish to prevent redefinition of method in subclasses
  - E.g. encryption classes won't let you define a subclass that prints out magic keys etc!
  - Author specifies method as `final`

## Re-iterating : concrete class

- Class that gets instantiated in program – you create instances (objects) of this class
  - All methods must be defined, compiler will complain that class is abstract if you forget to define a method

© nabg

## Using OO

© nabg

## Application specific hierarchy

- Does sometimes occur
- Not that common
- Revealed during initial “analysis” phase of application development (or possibly being identified during high level design)
- Classic example BankAccounts
  - BankAccount
    - SavingsAccount
    - CheckingAccount
    - Overdraft (Loan) account
    - Mortgage

© nabg

## Application specific hierarchy

- Analyst concerned with defining generic behaviours common to all accounts
  - getAccountID(), getBalance(), getTransactionHistory(), applyMonthlyCharges(), makeDepositTransaction(), makeWithdrawalTransaction(), ...
- Analyst develops rules on usage that distinguish different subclasses.
- Designer elaborates hierarchy identifying those behaviours that are common, and those that vary among subclasses; also must identify any subclass specific methods

© nabg

## Application specific hierarchy

- Sometimes the hierarchy may not exist in the real world application
  - it is more related to software implementation
- Remember the “Physics is Fun” example
  - Document (in Physics is Fun example) stored a variety of different kinds of data
  - Real world data objects quite different (no hierarchy)
  - Program world data objects quite similar (hierarchy)

© nabg

## Heuristic

- **Whenever** application appears to be using and storing a variety of **different kinds of data element**, **try** to see if creation of a data **class hierarchy** simplifies design.
- (Works for bank accounts, works for “physics is fun”, probably works for your application)
- *Heuristic doesn't apply if your data elements represent records from different tables in a database*

Heuristic = “rule of thumb”, a suggested action that usually works but doesn't have proven theoretical justification

© nabg

## Application specific hierarchy

- You will not invent them very often
- The ones that you invent will be simple shallow hierarchies
  - Interface
  - Abstract class with couple of final methods and a couple of default methods
  - Small number of concrete classes

© nabg

## Library hierarchy

- You don't invent these – you simply use classes and interfaces that exist.
- Java examples
  - Collection class hierarchy
  - AWT and Swing graphic library hierarchies
  - Others – e.g. encryption classes, data compression classes.

© nabg

## Collection classes

java.util  
**Interface Collection**

**All Known Subinterfaces:**  
[BeanContext](#), [BeanContextServices](#), [List](#), [Set](#), [SortedSet](#)

**All Known Implementing Classes:**  
[AbstractCollection](#), [AbstractList](#), [AbstractSet](#), [ArrayList](#), [BeanContextServicesSupport](#), [BeanContextSupport](#), [HashSet](#), [LinkedHashSet](#), [LinkedList](#), [TreeSet](#), [Vector](#)

---

public interface **Collection**

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The SDK does not provide any direct implementations of this interface: it provides implementations of more specific sub-interfaces like `Set` and `List`. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

© nabg

## Collection class hierarchy

- All collections have methods
  - Add another element (actually, “ensure its presence”)
  - Clear all current contents to get empty collection
  - Report if empty
  - Return an iterator that can be used to work through elements one by one
  - Get elements as array
  - Report size

© nabg

## Different kinds of collection

- interface `Set`
  - A collection that contains no duplicate elements
    - Add – only change set if element not already present
- interface `List`
  - An ordered collection (also known as a *sequence*).
    - `AbstractList`
      - `ArrayList`
      - `LinkedList`

© nabg

## Collections

- Write most of your code so that it works with the most abstract forms
  - Data members (local variables) are declared as instances of `Collection`
- Only where create collection do you decide whether using `LinkedList`, `ArrayList`, `Vector`, or other etc

© nabg

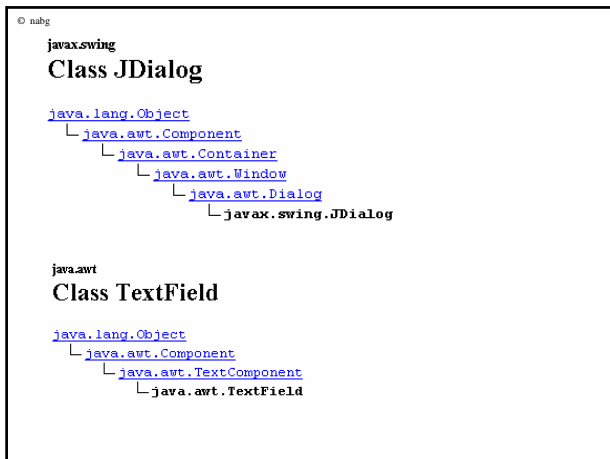
## Windows toolkits (AWT & Swing)

- Quite elaborate hierarchies

```

javax.swing
Class JButton
  java.lang.Object
    java.awt.Component
      java.awt.Container
        javax.swing.JComponent
          javax.swing.AbstractButton
            javax.swing.JButton

```



© nabh

## Using graphics classes

- Graphics part of Java program:
  - A complete framework
    - Own thread
    - Event dispatcher
      - Picks up “events” (from OS and Java runtime)
      - Dispatches to specific object that should handle that event
    - Schedules operations such as redrawing of windows
  - Framework code works in terms of “java.awt.Components”

© nabh

## java.awt.Component

- “A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user”
- A component can:
  - Return a bounding rectangle
  - Report whether it contains a point
  - Can be enabled or disabled (for input events)
  - Can claim focus (ability to accept events relating to keyboard input)
  - Etc
  - Etc
- But a component has no shape, no form, no appearance

© nabh

## A “concrete” Component : Button

```

java.awt
Class Button

java.lang.Object
├── java.awt.Component
│   └── java.awt.Button
└── java.awt.Button

java.awt.Button
– A component
– Linked to button as provided by runtime system on Windows, Gnome, Xwindows, ...
– Drawn by (C) code in runtime
– Accepts clicks, highlights when mouse over it etc

```

© nabh

## java.awt.Container

- “A generic Abstract Window Toolkit(AWT) container object is a component that can contain other AWT components. Components added to a container are tracked in a list.”
- So, classes like java.awt.Panel or javax.swing.JPanel – GUI elements used to group other elements in a complex display

© nabh

## Using graphics

- Ignore completely the framework and the hierarchy!
- Your program works in terms of the concrete classes
  - I’ll have a button here,
  - Here I’ll have a little panel holding three checkboxes
  - Here I want a text input field

## Using graphics

- Your code listens for events from your buttons
  - Do this now.
- Your code keeps references to elements like textfields
  - Ask element for input value when needed.

## Other standard hierarchies

- E.g.
  - java.security.cert
    - Certificate class
      - Concrete implementations
        - » Only X509Certificate provided by standard
        - » Others can be implemented

## OO?

- *Libraries with hierarchies?*
  - Not your job.
  - They have been standardized like awt and swing
- *Application-specific hierarchies*
  - Not that common
  - Usually not very elaborate
- *So where does all this OO stuff commonly arise?*

## So where will you use OO techniques?

- Common usage –
  - You define a new application specific subclass that fits into an existing application
- Examples – “servlets” and “enterprise Java beans” (from world of real Java applications)

## “Enterprise Java”

- Not part of CSCI213, you can get a limited introduction in CSCI399.
- “Servlet” is relatively simple example (and relates to the “... and the Internet” part of CSCI213)
- So brief explanation now to motivate your continued interest in OO.

## Servlet

- Servlets are a pure Java mechanism for handling WWW form-data entry and response.
- What do you need to do to handle data from a web form?
  - Manage lifecycle of some entity that accepts data and returns response (maybe create/destroy a process, maybe create/destroy an object in a threaded server)
  - Manage connections to databases
  - Have code that decodes (name,value) pairs with the data from the form that arrive in some encoded message
  - Process the data, and compose response
  - Return response to WWW client

## Most of the code is standard!

- You need essentially the same operations irrespective of application.
- C/C++ approach
  - Write it again, possibly using a little cut-&-paste editing to speed development.
- Modern approach:
  - Factor out as much of the standard code as possible
    - Framework code
    - Base class code

## Framework code

- Code like:
  - Manage lifecycle of request handlers
  - Manage database connections
  - Handle low-level communications,
    - unpack data
    - Compose header messages associated with responses
- Such code is completely application independent and can be made into a framework
  - Tomcat for servlets
  - JBoss, Sun's Application Server, IBM Websphere etc for more complex "enterprise beans"

## Base class (or hierarchy)

- Applications all have to do much the same
  - In case of Servlet, it is:  
*process these data pairs!*
- So
  - Define a base class (or hierarchy)
    - Implement parts where may have some default behaviour (so partially implemented abstract class)
  - Allow programmers to define application specific subclasses

```

javax.servlet.http
Class HttpServlet

java.lang.Object
|-- javax.servlet.GenericServlet
    |-- javax.servlet.http.HttpServlet

All Implemented Interfaces:
    Serializable, Servlet, ServletConfig

public abstract class HttpServlet
extends GenericServlet
implements Serializable

Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.

```

## Servlet

- public interface **Servlet**
  - Defines methods that all servlets must implement.
  - A servlet is a small Java program that runs within a Web server.
  - Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

## GenericServlet

- public abstract class **GenericServlet**
  - extends Object
  - implements Servlet, ServletConfig, Serializable
- Defines a generic, protocol-independent servlet.
- **GenericServlet** makes writing servlets easier.
  - It provides simple versions of the lifecycle methods `init` and `destroy` and of the methods in the `ServletConfig` interface.
  - `GenericServlet` also implements the `log` method, declared in the `ServletContext` interface.

© nabg

## HttpServlet

- **public abstract class HttpServlet**
  - extends GenericServlet
  - implements Serializable
- Provides an abstract class to be subclassed to create an HTTP servlet suitable for a Web site.
- A subclass of **HttpServlet** must override at least one method, usually one of these:
  - **doGet**, if the servlet supports HTTP GET requests
  - **doPost**, for HTTP POST requests
  - **doPut**, for HTTP PUT requests
  - **doDelete**, for HTTP DELETE requests
  - **init** and **destroy**, to manage resources that are held for the life of the servlet
  - **getServletInfo**, which the servlet uses to provide information about itself

© nabg

## MyServlet

```

java.lang.Object
|
+-- javax.servlet.GenericServlet
|
+-- javax.servlet.http.HttpServlet
|
public class MyServlet
    extends HttpServlet
    {
        public void doGet(...) throws ...
        { ... }
        ...
    }
  
```

© nabg

## public class MyServlet

- **public class MyServlet extends HttpServlet**
  - Application specific
  - **doGet** method
    - Returns HTML text specifying a data entry form
  - **doPost** method
    - Consumes data that are submitted via the form, generates a response page

© nabg

## Application development for Servlets

- Don't bother about
  - Lifecycle management
  - Access control
  - Session maintenance
- Tomcat server and servlet framework code handle those
- Simply implement a subclass
  - Override default methods if necessary
  - Define effective methods for any that are abstract

© nabg

## Use of OO

- This style of code is where you will most likely encounter OO concepts
- You create application specific subclasses that will be run by pre-tested, robust frameworks
- Your concern is purely the business methods
- The framework provides all the infrastructure – difficult, error-prone code like lifecycle management

© nabg

## finished

But wait, ... there's more

# JavaDoc

## JavaDoc

- By now you should all have learnt to love browsing around in the JavaDoc documentation that describes the classes in the Java standard packages



## JavaDoc – tools etc

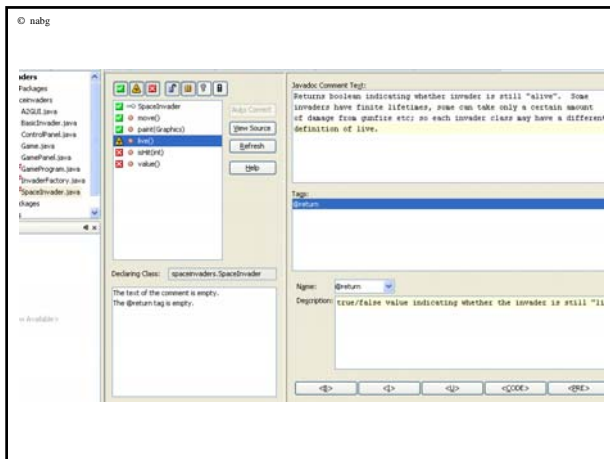
- Java system comes with a tool that generates HTML style documentation of this form.
  - javadoc
    - command-line program like javac and java
    - bit painful to use – arguments etc
- System relies on a scheme for adding comments in a defined style

## JavaDoc comments ...

```
/**  
 * Returns an Image object with data loaded from a file in the  
 * "user directory" (typically the directory where the Java .class file  
 * is located)  
 * Pauses the program until image is loaded.  
 * @param name Name of file with image  
 * @return Image object  
 */  
public static Image getImage(String name) {
```

## JavaDoc comments

- Somewhat tiresome to edit in by hand and get correct ...
- NetBeans automates
  - Addition of comments in correct format
  - Generation of final JavaDoc



Use JavaDoc ...

- Assignment 2 (and possibly also later assignments) will require some JavaDoc documentation in your submission.
- Get into habit of adding JavaDoc comments as you implement the classes
  - It is easy in NetBeans
  - It helps you in the long run