

## The Building Blocks

- ◆ Classes:
  - ◆ Classes have variables and methods.
  - ◆ No global variables, nor global functions.
  - ◆ All methods are defined inside classes (except native methods)
- ◆ Java class library, over 1,800 classes:
  - ◆ GUI, graphics, image, audio
  - ◆ I/O
  - ◆ Networking
  - ◆ Utilities: set, list, hash table

## Organization of Java Programs

Java provides mechanisms to organize large-scale programs in a logical and maintainable fashion.

- ◆ Class --- highly cohesive functionalities
- ◆ File --- one class or more closely related classes
- ◆ Package --- a collection of related classes or packages

The Java class library is organized into a number of packages:

- ◆ `java.awt` --- GUI
- ◆ `java.io` --- I/O
- ◆ `java.util` --- utilities
- ◆ `java.applet` --- applet
- ◆ `java.net` --- networking

## A Simple Applet --- Digital Clock

```
import java.awt.*;
import java.util.Calendar;

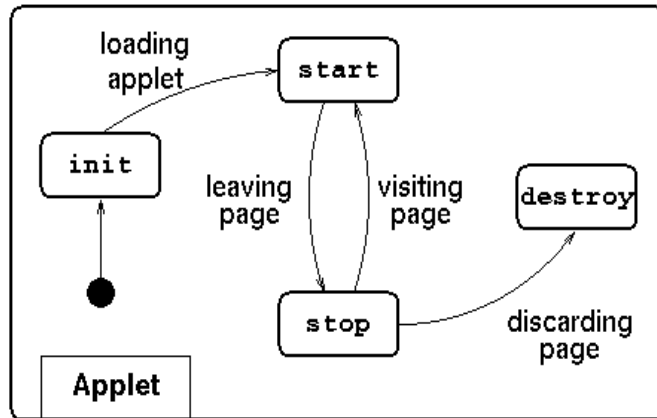
public class DigitalColok
    extends java.applet.Applet
    implements Runnable {
    <Fields>
    <Methods>
}
```

- ◆ The `import` clause is not necessary to use the library. It is only a convenience.
- ◆ An applet must be a subclass of `java.applet.Applet`.

## The Applet Methods

- ◆ `public void init(){...}`  
invoked when the applet is loaded initially
- ◆ `public void start(){...}`  
invoked when entering the web page that contains the applet
- ◆ `public void stop(){...}`  
invoked when leaving the web page that contains the applet
- ◆ `public void run(){...}`  
run the applet, i.e., the main driver of the applet
- ◆ `public void paint(Graphics g){...}`  
paint the picture

## The Life-Cycle of An Applet



## Fields and Initialization

```
protected Thread clockThread = null;
protected Font font =
    new Font("Monospaced", Font.BOLD, 48);
protected Color color = Color.green;
```

By default all class fields are automatically initialized to their *default values*, usually 0 or null.

## The start() and stop() Methods

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this);
        clockThread.start();
    }
}

public void stop() {
    clockThread = null;
}
```

Start and stop the thread.  
Stopped threads will not consume CPU time.

## The run() Method

```
public void run() {
    while (Thread.currentThread()
        == clockThread) {
        repaint();
        try {
            Thread.currentThread().sleep(1000);
        } catch (InterruptedException e){}
    }
}
```

In each iteration, `repaint()` is invoked, then sleep 1 second.  
`Sleep()` must be invoked inside the try block.

## The paint ( ) Method

```
public void paint(Graphics g) {
    Calendar calendar = Calendar.getInstance();
    int hour = calendar.get(Calendar.HOUR_OF_DAY);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);
    g.setFont(font);
    g.setColor(color);
    g.drawString(hour +
        ":" + minute / 10 + minute % 10 +
        ":" + second / 10 + second % 10,
        10, 60);
}
```

## Who Calls run ( ) And paint ( )?

- clockThread.start ( ) calls DigitalClock.run ( )
- DigitalClock.repaint ( ) calls DigitalClock.paint ( )
- The paint ( ) method is usually not called directly.

## Drawing Strings

```
g.drawString("A Sample String", x, y)
```



## HTML Source

```
<!--DigitalClockDemo.html-->
<html>
  <head>
    <title>Digital Clock Applet</title>
  </head>
  <body bgcolor=white>
    <h1>The Digital Clock Applet</h1><p>
    <applet code=DigitalClock.class
      width=250 height=80>
    </applet>
    <p><hr>
    <a href=DigitalClock.java>The source</a>
  </body>
</html>
```

## The Blinking Text Applet

```
import java.awt.*;
import java.util.StringTokenizer;

public class Blink
    extends java.applet.Applet
    implements Runnable {
    // variables
    public void init() { ... }
    public void paint(Graphics g) { ... }
    public void start() { ... }
    public void stop() { ... }
    public void run() { ... }
}
```

## The Blinking Text Applet (cont'd)

### Fields and initialization

```
protected Thread blinkThread;
protected String blinkingtext;
protected Font font;
protected int speed;

public void init() {
    font = new Font("Serif",
                    Font.PLAIN, 24);
    speed = 400;
    blinkingtext = "This is a simple blinking" +
                   "text applet for demonstration in SE 450";
}
```

## Blinking Text --- paint ( ) Method

```
public void paint(Graphics g) {
    int x = 0, y = font.getSize(), space;
    int red = (int)(Math.random() * 50);
    int green = (int)(Math.random() * 50);
    int blue = (int)(Math.random() * 256);
    Dimension d = getSize();

    g.setFont(font);
    FontMetrics fm = g.getFontMetrics();
    space = fm.stringWidth(" ");
```

`Math.random()` generates floating-point random numbers in [0.0, 1.0)

## Blinking Text --- paint ( ) Method (cont'd)

```
for (StringTokenizer t =
     new StringTokenizer(blinkingtext);
     t.hasMoreTokens(); ) {
    String word = t.nextToken();
    int w = fm.stringWidth(word) + space;
    if (x + w > d.width) {
        x = 0;
        y += font.getSize();
    }
```

`StringTokenizer` breaks the `blinkingtext` into words, i.e., tokens.

## Blinking Text --- paint ( ) Method (cont'd)

```
    if (Math.random() < 0.5) {
        g.setColor(new java.awt.Color(
            (red + y * 30) % 256,
            (green + x / 3) % 256,
            blue));
    } else {
        g.setColor(getBackground());
    }
    g.drawString(word, x, y);
    x += w;
}
```

Drawing some of the words in the background color and the other words in random color.

## The java.awt.Color Class

- Instances of the Color class represent colors.

```
new Color(r, g, b)
```

where *r*, *g*, *b* are the values of the red, green, and blue components, respectively. They are in the in the range of 0 to 255.

- Some common colors are predefined as constants.

black	gray	orange	yellow
blue	green	pink	
cyan	lightGray	red	
darkGray	magenta	white	

## The java.awt.Font Class

- Fonts are specified with three attributes:
  - *font name*:  
Serif Sans-serif Monospaced Dialog DialogInput  
TimesRoman Helvetica Courier Dialog
  - *font style*:  
PLAIN BOLD ITALIC  
Styles can be combined: Font.BOLD | Font.ITALIC
  - *font size*: a positive integer
- A font can be created as follows:  

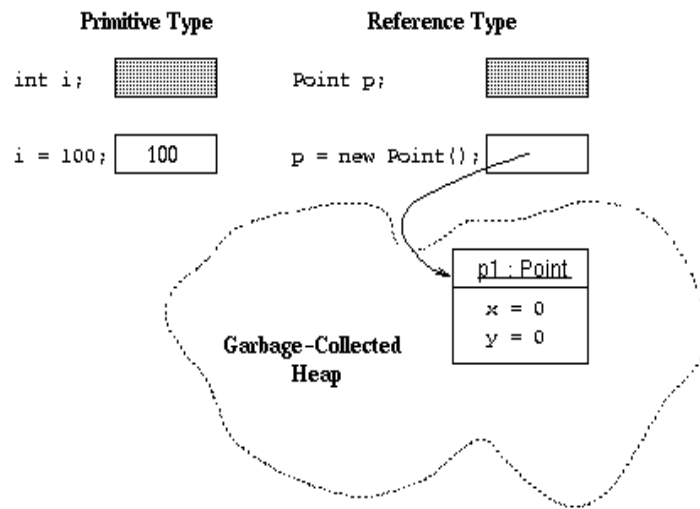
```
new Font(name, style, size)
```

## Java Data Types

- Primitive Types:

boolean	true and false
char	16-bit Unicode
byte	8-bit integer
short/int/long	16/32/64-bit integer
float/double	32/64-bit floating-point (IEEE-754)
- Reference Types: objects and arrays
  - Every class is a subclass of Object.
  - null --- a special value, null reference
- Java expressions and control flows are very similar to those of C/C++.

# Reference Type and Garbage Collection



# Java Character Type

- Internationalization
  - 16-bit Unicode 1.1.5 character.
  - ASCII is a subset of Unicode --- ISO-8859 (Latin-1)
  - Escape sequence:
    - `\uhhhh`: hex-decimal code, e.g. `\u000A`
    - `\ddd`: octal code, e.g. `\040`
    - `\n`, `\t`, `\b`, `\r`, `\f`, `\\`, `\'`, `\"`.
- Java programs are also in Unicode.
- Unicode standard: <http://www.unicode.org>

# Java Arrays

- Arrays are first-class objects.
- Arrays are always bound-checked.
- Array index starts from 0.

```
int[] ia = new int[3];
int ia[] = new int[3];
int[] ia = { 1, 2, 3};

float[][] mat = new float[4][4];

for (int y = 0; y < mat.length; y++) {
    for (int x = 0; x < mat[y].length; x++)
        mat[y][x] = 0.0;
}
```

# Java String

- Strings are first-class objects.
- Strings are not arrays of char's.
- String index starts from 0.
- String constant  
"A String constant"
- String concatenation  
`s1+s2`     `s1+=s2`
- `s.length()`  
the length of a string s.
- `s.charAt(i)`  
character at position i.

## Class Declaration

```
[ClassModifiers] class ClassName
  [extends SuperClass]
  [implements Interface1, Interface2 ...] {
    ClassMemberDeclarations
  }
```

## Class Modifiers

`public`  
Accessible everywhere. One public class allowed per file. The file must be named *ClassName*.java

`<empty>`  
Accessible within the current class package.

`abstract`  
A class that contains abstract methods

`final`  
No subclasses

## Method And Field Declaration

```
[MethodModifiers] Type Name ( [ParameterList] ) {
  Statements
}
```

```
[FieldModifiers] Type FieldName1 [= Initializer1] ,
  FieldName2 [= Initializer2] ... ;
```

## Method and Field Modifiers

- For both methods and fields  
public protected private  
static final
- For methods only  
abstract synchronized native
- For fields only  
volatile transient

## Accessibility of Class Members.

	public	protected	package	private
The class itself	Yes	Yes	Yes	Yes
Classes in the same package	Yes	Yes	Yes	No
Subclasses in a different package	Yes	Yes	No	No
Non-subclasses in a different package	Yes	No	No	No

## Class Declaration Example

```
public class Point {
    public int x, y;

    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}

Point point1; // Point not created
Point point2 = new Point();
point1 = point2;
point1 = null;
```

`x` and `y` are initialized to their *default initial values*.

## Explicit Initializer

```
public class Point {
    public int x = 0, y = 0;

    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}

Point point1 = new Point(); // (0,0)
```

## Constructors

```
public class Point {
    public int x, y;

    public Point() { // no-arg
        x = 0; y = 0;
    }

    public Point(int x0, int y0) {
        x = x0; y = y0;
    }
}

Point point1 = new Point(); // no-arg
Point point2 = new Point(20, 20);
```

- ◆ Constructors are invoked after default initial values are assigned.
- ◆ No-arg constructor is provided as a default when no other constructors are provided.

## Variable, Object, Class, Type

Variables have types, objects have classes.

- ♦ A *variable* is a storage location and has an associated *type*.
- ♦ An *object* is a *instance* of a *class* or an array.
- ♦ The type of a variable is determined at compilation time.
- ♦ The class of an object is determined at run time.
- ♦ A variables in Java can be of:
  - ♦ primitive type --- hold exact value
  - ♦ reference type --- hold pointers to objects
    - ♦ null reference: an invalid object
    - ♦ object reference: an object whose class is *assignment compatible* with the type of the variable.

## Object Reference: `this`

You can use `this` inside a method,

- ♦ It refers to the current object on which the method is invoked.
- ♦ It's commonly used to pass the object itself as a parameter

```
aList.insert(this);
```

- ♦ It can also be used to access hidden variables:

```
public class Point {
    public int x, y;
    public Point(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

## Static Variables

*Static variable* or *fields*: one per class, rather than one per object.

- ♦ Static variables are also known as *class variables*.
- ♦ Non-static variables are also known as *instance variables*.

```
public class IDCard {
    public long id;
    protected static long nextID = 0;

    public IDCard() {
        id = nextID++;
    }
}
```

## Static Methods

A *static method*

- ♦ can only access static variables and invoke other static methods;
- ♦ can not use `this` reference.

```
public class IDCard {
    public long id;
    protected static long nextID = 0;
    ...
    public static void skipID() {
        nextID++;
    }
}
```

## Invoking Methods

- ◆ Non-static methods must be invoked through an object reference:

```
object_reference.method(parameters)
```

- ◆ Static methods can be invoked through an object reference or the class name:

```
class_name.method(parameters)
```

So, you can do either of the following:

```
IDCard.skipID(); // the preferred way
```

```
IDCard mycard = new IDCard();  
mycard.skipID();
```

## final Variables

Final variables are *named constants*.

```
public class CircleStuff {  
    static final double pi=3.1416;  
}
```

Final variables are also static.

## The toString() Method

The toString() method converts objects to strings.

```
public class Point {  
    public int x, y;  
    //...  
    String toString() {  
        return "(" + x + "," + y + " ";  
    }  
}
```

Then, you can do

```
Point p = new Point(10,20);  
System.out.println("A point at " + p);  
// Output: A point at (10,20)
```