

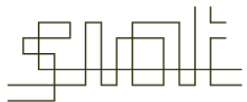
Introduction and Programming Basics

Lecture 1

IAT-265, Summer 2024

School of Interactive Arts and Technology

Slides based on materials from IAT265 offerings
by Eric Yang and Marek Hatala



SCHOOL OF INTERACTIVE
ARTS + TECHNOLOGY

SCHOOL OF INTERACTIVE ARTS + TECHNOLOGY [SIAT] | WWW.SIAT.SFU.CA

Teaching team

■ Instructor: Eric Yang (yyang1@sfu.ca)

— Office hours:

- Friday 11:30am - 12:30pm
- Over Zoom (To attend, please email one day ahead to book appointments)

■ TAs:

— Arash Ahrar (arash_ahrar@sfu.ca)

- Monday TBA
- ### — Location: TBA

Topics

- About the course
- Java: an OOP language
- Two types of Java program: Application and Applet
- File structures of Java program
- Method, parameter, argument
- Data types and Variables

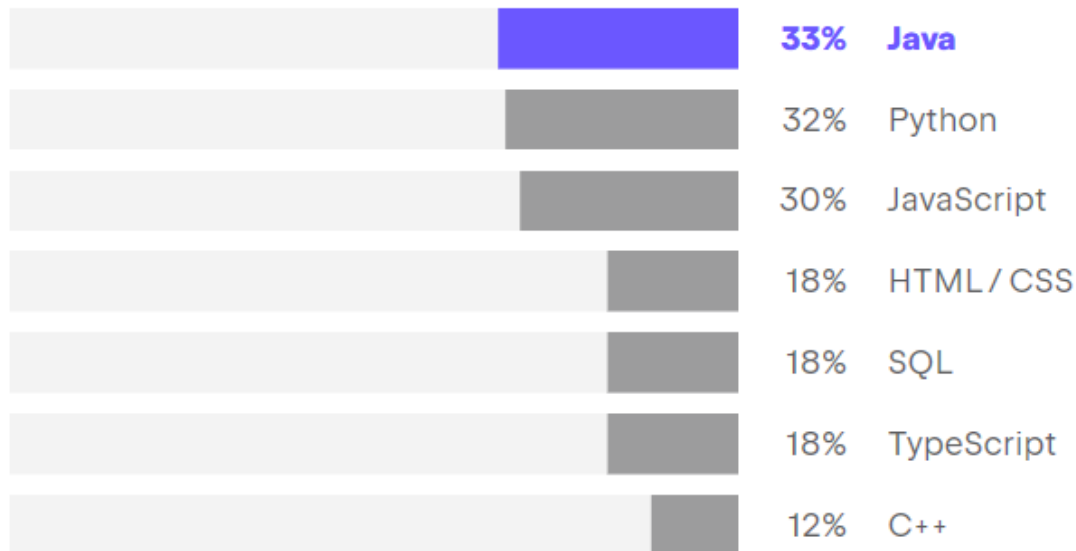
About the course

- Review and **consolidate** concepts such as *variables, data types, conditionals, ArrayLists, loops, OOP, event-driven programming* etc, with the most popular primary programming language – *Java*
- Introduce **new** concepts: e.g. *data hiding, inner class, abstract class, interface, event listener, timer, data structures, search, sorting, etc.*
- Learn to **design** multimedia programs using *concept /software design, pseudo-coding-process(PPP), code refactoring*, and *Design Patterns*
- Learn to **implement** per the design using both **native** and **external libraries**

State of Language Ecosystem 2023

What are your primary programming, scripting, and markup languages?

Choose no more than three languages.



Is it still worth to learn Java?



Dave Voorhis

Software entrepreneur, engineer, and educator for 35+ years. · 8mo



Is it worth it to learn Java in 2022 despite having Python and Go type programming languages in the market?

In the enterprise backend space — which is a *huge* space — Java rules. Go barely gets a look in, whilst Python is a handy tool for scripting and peripheral tasks.

But Java handles the bulk of line-of-business heavy lifting.

In the mobile space — which is a *big* space — Java rules on Android. Go and Python, no, not at all.

It's *definitely* worth learning Java, but what's even *more* worthwhile is learning computer science.

Learn it, and you'll learn that programming languages are implementations of fundamental principles that don't change, whilst programming languages come and go.

Learn those fundamental principles, and it won't matter what languages are in the market — you'll be able to pick up any of them as needed and use them productively.

Connection between 167 and 265

- The setting of the two courses are followed with the **Spiral Model** – *iterative* and *incremental*
 - Break complex concepts (like OOP) into components at different levels to conquer with iterations
 - Cover the basics of *OOP* and *Event Driven Programming* in the 1st iteration (done in 167)
 - Revisit those fundamental concepts in the next iteration to consolidate (done in 265)
 - Add new increments at higher level to strengthen (done in 265)

Design,
Implement,
test

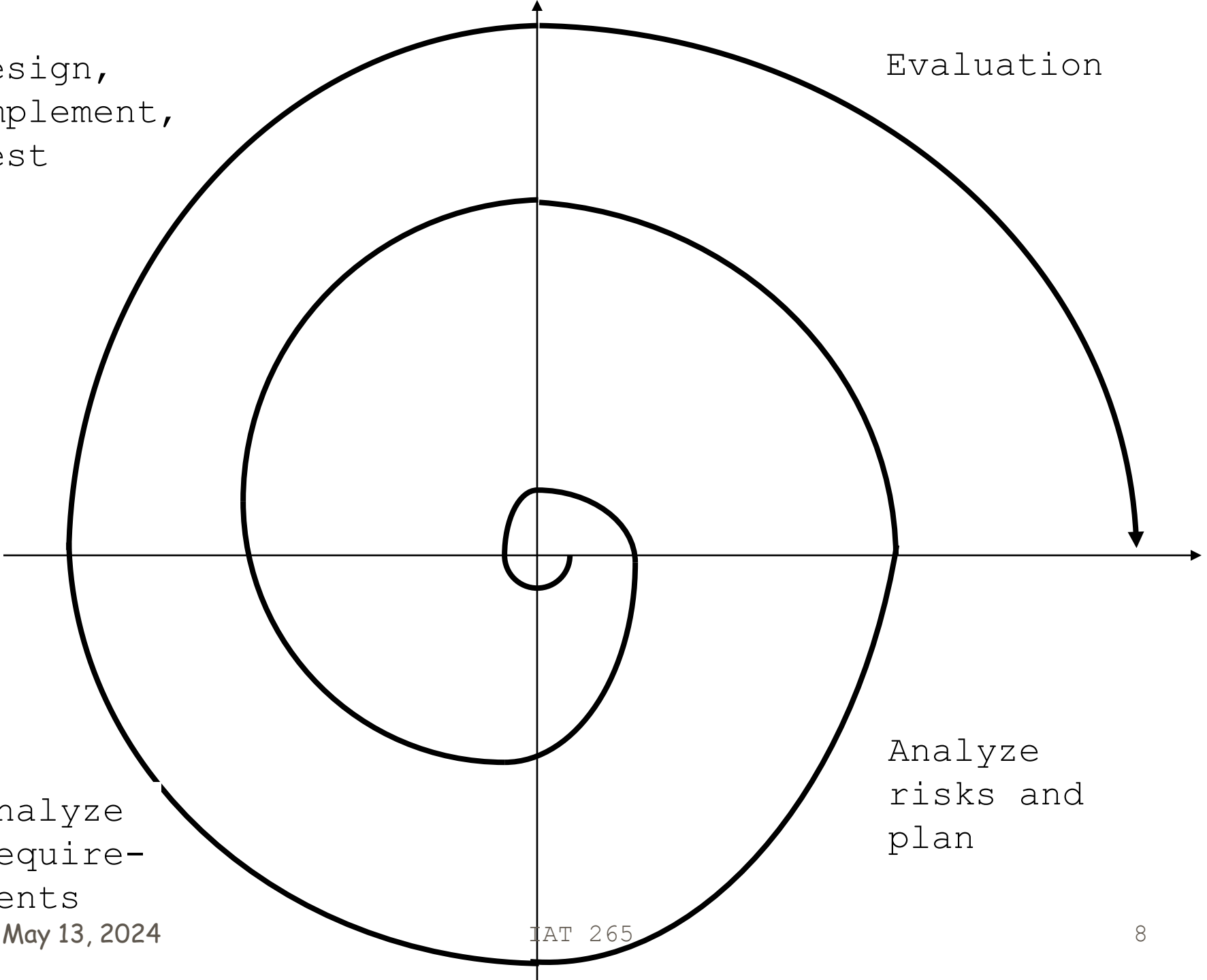
Evaluation

Analyze
risks and
plan

Analyze
require-
ments

May 13, 2024

IAT 265



Building on IAT167

- You need to know what you have learned in the prereq course
 - How?
- Mandatory *Programming Proficiency quiz* and *Self-efficacy* survey in Canvas:
 - Quiz: simple programming task, 60 minutes
 - Due date: Fri May 10, midnight
 - Survey: 10 questions, no time limit
 - Due date: Sat May 11, midnight

Syllabus – anatomy along concepts

■ Computing concepts

- Advanced OOP concepts
 - **Data hiding**
 - Data accessing/mutating
 - **Abstract class**
 - **Java interface**
- Event driven programming
- **Recursion**

■ Working with libraries

- Java native library
- External libraries (e.g. Processing, Minim)

■ Multimedia creation

- Shapes
- **Transformations**
- **Graphics/Images**
- **Text**
- **Interactions**

■ Other advanced concepts

- **Code-Refactoring**, and **Design Patterns**
- Basic **data structures** and **algorithms**

Syllabus – anatomy along techniques

- Concept design
 - **Problem** to be solved
 - **Data** involved
 - **Functionalities** to have
 - **Constraints** presented
- Software design
 - How to **represent** data (classes, their relations, variables, constants)
 - How to **achieve** functionalities (methods and execution flow)
 - How to **handle** constraints
- Software implementation
 - Coding as per software design using certain programming languages
- Motion and animation
 - Timer based motions
 - Micro-animations for characters and environment
- **Interactions**
 - Mouse interactions
 - Keyboard interactions
 - GUI interactions
 - **Image based interactions**
 - Object interactions: **collision detection** and **avoidance**
- **FSM** for creation of AI behaviors and advanced interactions
 - Use number to represent different states
 - Use conditionals to manage flow control
- Generating special shapes
 - **Perlin noise** for procedural textures
 - Recursion for **fractals**
- Sound effects with native and external libraries
 - Java native sound library (limited functionality)
 - Minim library (more powerful)

Goals for the course

- Gain programming **knowledge** and **skills**
 - Analyze requirements
 - Solve problem algorithmically
 - Implement (write code to fulfill the requirements)
- Instructor, lectures, and quizzes/exams are primarily to help you gain the **knowledge** and how the knowledge can be applied in **context**
- TAs, Tutorials and assignments focus more on building up the **skills** based on the knowledge and context
- Apparently you count on both to work interactively to allow you to gain the **capability** to solve any problem using software approach

Grading

- Lecture participation: 5% (examining concept/technique understanding within lectures)
- Lab participation & challenges: 10% (skill building weekly)
- 4 Assignments: 38%
 - Assignment Phase 1 (Concept design + PPP or Design + Coding)
 - Assignment Phase 2 (Implementation – coding)
 - Assignment Assessment (examining concept understanding)
- 3 Quizzes: 15% (examining concept understanding periodically)
- Final Exam: 30% (examining concept understanding holistically)
- Research participation: 2%

New SIAT Grading Scale

Name:	Range:	
A+	100 %	to 95.0%
A	< 95.0 %	to 90.0%
A-	< 90.0 %	to 85.0%
B+	< 85.0 %	to 80.0%
B	< 80.0 %	to 76.0%
B-	< 76.0 %	to 72.0%
C+	< 72.0 %	to 68.0%
C	< 68.0 %	to 64.0%
C-	< 64.0 %	to 60.0%
D	< 60.0 %	to 50.0%
F	< 50.0 %	to 0.0%

About Lecture Participation

- Mainly done with *mini-quizzes* in *Canvas* using either *phone or laptop*
 - Each of the *mini-quizzes* will examine the important **concepts** covered in the **current lecture** using multiple choice questions
 - You get credits for both participation and performance
 - The dates for these quizzes are **not predetermined** but based on the time availability
 - So make sure to **attend each of the lectures** and bring with you either your phone or laptop!!

Some Other Admin Issues

- All class announcements will be done via our class email list:
 - iat-265@sfu.ca
 - It is your responsibility to check the email daily for any announcement
- Lab switching:
 - Possible **only if** you formally complete the lab switch in SIMS via SIAT advising
- Grading Disputes
 - For any **assignment** grading issue, please contact your TA (the grader) **in writing** for review within **7 days** after the grades are released
 - Appeal to instructor **in writing** only when the issue is unable to resolve between you and your TA and provide appropriate reasons within **10 days** after the grades are released as per SFU regulations

Requirements and Expectations

- **Attend** all lectures and labs
 - Slides and coding demos are put in Canvas before each class
 - There are materials I talk about that are not in the slides. You are expected to know it for the quizzes/exams as well – take notes as part of your review references
- Do all the **ASSIGNMENTS!!**
- Do reviews on a **WEEKLY** basis
 - Read **slides** and **readings** (provided via Canvas), and try to understand the basic concepts
 - **Attend instructor/TA office hours for clarification**
- **Practice, Practice, Practice!**
 - Programming courses are always challenging and need practice

...on how to get a good grade

- Do the readings and review slides after the lecture AND before the labs (desirably)
- Run the demos yourself as you are reading through the slides. Try to do some modifications on your own
- Be fully engaged in the lab
- Do each of the lab challenges
- If you are unclear on concepts at that point, re-read the sections, and if still missing the point **attend the office hours**
- Read lectures slides/your notes/readings before the mini-quizzes/quizzes. Review tutorials and your own assignments for major concepts and techniques

NO Plagiarism

- Absolutely **NO** plagiarism and cheating will be tolerated
 - Don't copy previous students' work of the course
 - Don't copy others and don't let others copy you
 - The **minimum penalty** for **all parties** involved will be **negative of the assignment's total marks** (e.g. if the assignment is worth 20 pts, then **-20pts** would apply)!!
- We keep the right to **report to the School director and the Registrar** for record keeping per SFU Policy S10.1, Appendix 3

Note: avoid potential plagiarism behavior in doing assignments

- One type of **Plagiarism behavior** defined by SFU Code of Academic Integrity (linked in the Syllabus):
 - “ii. copying all or part of an essay or other assignment from an author or other person, including a **tutor** or **student mentor**, and presenting the material as the student’s original work”
- Based on this, our criteria for determining **non-plagiarism** when referencing lab tutorials/lecture demos is as follows:
 - The **framework** of your assignments must be **yours**: including names of classes, fields, and method names, visuals etc.
 - You can **draw references** from lab tutorials and lecture demos **for solving problems within your own framework**
- Cases that will be deemed as **plagiarism** for the followings:
 - Use a **lab tutorial**, **lecture demo**, or **any other’s work** (peer’s, previous student’s etc.) **as a whole** to be yours, and do **minor modifications on top**
 - The code directly taken from another source is **more than 50%**

About taking advantage of LLM like ChatGPT etc.

- While AI-based coding assistants could help with code completion and code generation, the fundamentals of programming remain: the ability to read and reason about your own and other's code, and understanding how the code you write fits into the larger system
- *GitHub Copilot* can offer suggestions as you code. *ChatGPT* and Google's *Bard* act more like conversational AI programmers and can be used to answer questions about APIs or generate code snippets
 - *A human coder is still the one who has to figure out the structure of a piece of code, and right abstractions around which to organize it, and the requirements for different interfaces* – Solar-Lezama, COO of MIT CS and AI Lab

About taking advantage of LLM like ChatGPT etc.

- Be critical and understand the risks
 - Software engineers should be critical of the outputs of large language models, as they tend to hallucinate and produce inaccurate or incorrect code
- *We should be making AI a copilot – not the autopilot – for learning* – J. Chang, Stanford University
- We encourage you to explore and use these AI tools as assistant to help clarify API questions and even generate part of the code for your assignments
 - Under the condition that you have *established your own structure* with the *right abstractions*, and understand the how the *code generated fits into your structure**

*The rule still holds: *The code directly taken from another source is **more than 50%***

■ Any questions?

What is Multimedia?

- Applications that use multiple modalities ... of **text, images, drawings (graphics), animation, video, sound, and interactivity**
 - Z. Li & M. Drew *Fundamentals of Multimedia*
- **Multimedia** (Lat. Multum + Medium) is media that uses multiple forms of information content and information processing (e.g. **text, audio, graphics, animation, video, interactivity**) to inform or entertain the (user) audience
 - From *Wikipedia*, the online encyclopedia

Typical Multimedia Applications

- *World Wide Web*
- *Video Games*
- *Virtual/Augment Reality* (aka Immersive Multimedia)
- *CGIs* (in film making)
- *Video teleconferencing*
- *Multimedia courseware*
- *Scientific visualization*
- *Simulations* (of systems, processes etc.)
- ...

Some Multimedia Apps

- Created by students of the class ...
 - Watch the demos

Introduction to Java

- A fully object-oriented language
- Types of Java program
- Java language and Eclipse IDE
- Basics of Java Syntax
- File structure of Java apps
- Java system library and APIs

Java is Fully Object-Oriented

- We live in a world full of objects
 - Images, cars, remote controls, televisions, employees, students, ...
- The older languages are procedural
- OOP languages have the added capability to encapsulate objects' properties and functions into one container – *object*
 - A template for creating *objects* is called *Class*

Object Oriented vs. Procedural Languages

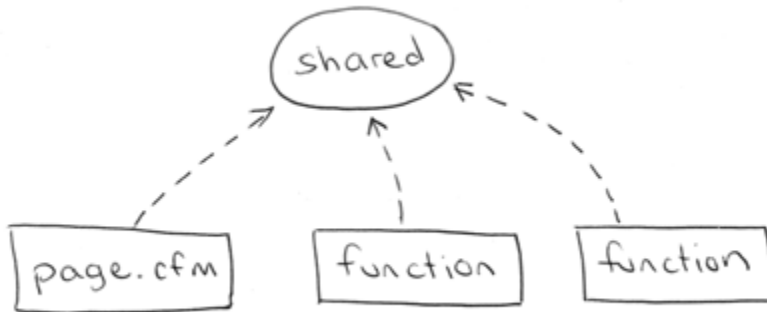
Procedural (e.g. C)

- We create some data representing an image
- We write a *procedure* (*i.e. function*) that can accept the data and draw the image

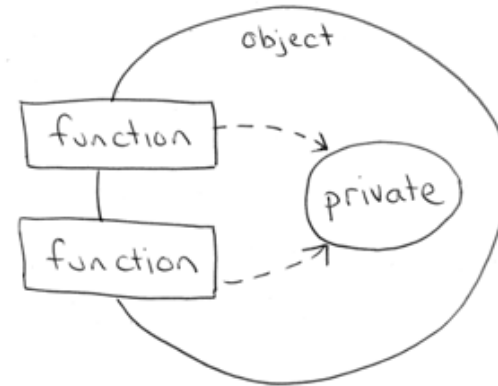
Object Oriented (e.g. Java)

- We create an object that contains image data AND a function to draw it
- The data and the function (ability to draw) are in ONE "container" – the object

To generalize ...



- For **procedural**, functions have no intrinsic relationship with the data they operate on
- Data accessed in this way is considered "global" or "shared" data



- In OOP, the data and related functions are bundled together into an "object"
- Ideally, the data inside an object can only be manipulated by calling the object's functions

Two types of Java Programs

■ Java Application:

- Standalone program, runs on its own

■ Java Applet:

- small program, embedded inside web applications (becoming obsolete)


Our First Java Application

■ HelloWorld.java

```
/*  
 * Hello World  
 * The classic first program  
 */  
public class HelloWorld {  
    public static void main(String args[]) {  
        //print a message to the console  
        System.out.println("Hello, world!");  
    }  
}
```

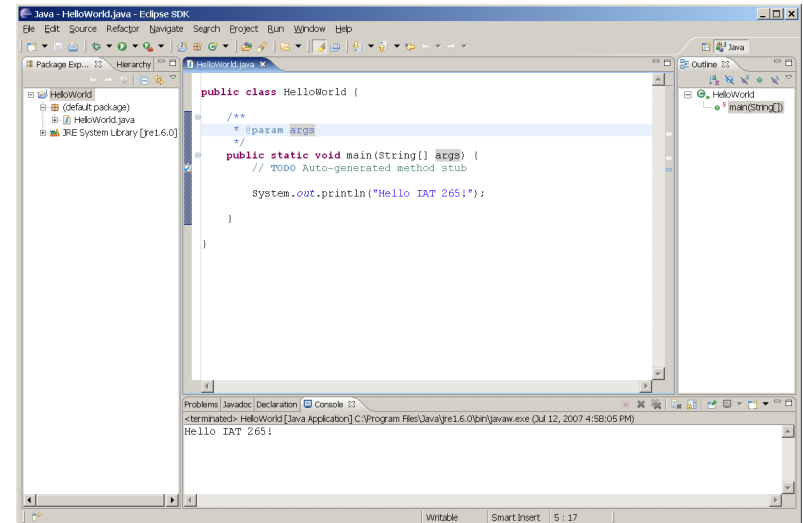
- Please note: for Java, the **source file name** must match the **class name** (case sensitive) for it to compile

We use Eclipse 2024-03 R

- Where *compile* is done automatically whenever you save your file
- You run your program by selecting "*Run as*" on the menu or the  button for running

Please note: you must have **Java SDK** installed before you can run Eclipse

- We are using **Java SDK 17** - make sure you **DON'T** install **Java 21** – your code might run into some incompatible issue if run on machines with lower version!!



Basics of Java Syntax

```
public class HelloWorld {
```

- You must define a class in every Java program
- **public** – keyword for access control that allows all other classes in your program to access
 - **private** – can only be accessed within the class where the member (field or method) is defined
- As a convention, **class** name starts with a **Capital** letter (vs. **variable** and **method** names - both start with a **lowercase** letter), and then goes **CamelCase** for each additional word

main method as the driver for Java Application

```
public static void main(String args[]) {
```

- **main** method drives Java applications, which is the first place the JRE visits when running your application
- **static** – keyword that specifies that the member belongs to the class instead of a specific instance

Including Comments

- Two basic types of comments in Java
 - in-line and block comments
- **in-line comments**: make a note about a particular line of code, e.g.

```
//printing a message to the console  
System.out.println("Hello, World!");
```

- **block comments**: normally used to explain a class or method on top

```
/*  
 * Hello World  
 * The classic first program  
 */
```

Call Java API Methods

```
System.out.println("Hello, world!");
```

- *System* class has a **static** field: *out* – an object of *PrintStream*, which has overloaded methods *for printing*
- A call to a method that handles your system's standard output (aka console)
 - Refer to Java API documentation for more detail:
<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

Review:

Define your own Method

- **Methods** are reusable commands
 - Like a tool or device that does work for you
 - Let you reuse code without typing it over and over
- You can define your own methods by providing:
 - *method signature* + { *method body* }

Review: Method Signature

- In computer programming, a method is identified by its **method signature**
 - Just method name is not enough to do this, as method *overloading* is allowed in Java
- Method signature: **return_type** + **method name** + **parameters** :
 - `void main(String args[])`
 - `void println(String x)`
 - `int max(int a, int b)`
 - `double random()`
- * Strictly speaking, method signature doesn't include the *return type*. In this course though, we'd like to include the *return type*, so that we can learn what type of *value* we can expect by simply looking at its signature

Define methods with parameters

```
public class HelloSomeone {  
    public static void main(String[] args) {  
  
        //accept someone's first & last name as arguments  
        sayHello("Eric", "Yang");  
        sayHello("Arash", "Ahrar");  
        sayHello("Kimia", "Aghaei");  
  
    }  
  
    public static void sayHello (String firstNm, String lastNm) {  
  
        System.out.println("Hello " + firstNm + " " + lastNm +  
            " !");  
    }  
  
}
```


Review:

Parameters vs Arguments

- In Java, if a method is defined with parameters, you must pass an argument for each of the parameters when you call it

Parameters: place holders for values

sayHello (**String firstNm, String lastNm**)

sayHello ("**Eric**", "**Yang**")

Arguments: actual values passed in

File Structure of Java Applications

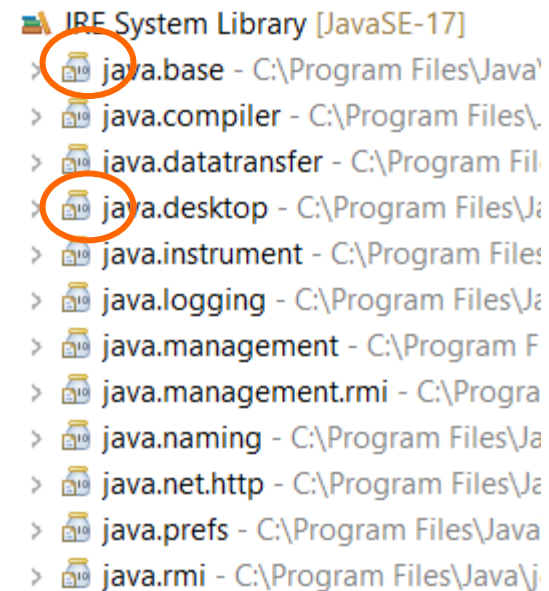
- All Java files are sitting in a **package**
 - You can create named packages explicitly. Eg:
 - ***package com.game.enemy;***
 - Otherwise a **default package** will be generated by Eclipse for you to use (**NOT** a good practice, but we'll start with it)
- Each file can hold only one **public** class
- In Java, the **source file's name** must match exactly (case sensitive) the **name of the public class** it holds to compile properly

File Structure of Java Applications (1)

- If one class in a package wants to access another class in a different package, it must import that class first. Eg:
 - *import com.game.gameTokens.EnemyAmmo;*
 - *import java.awt.event.KeyEvent;*
 - *import java.awt.Graphics2D;*
 - *import java.swing.*;*
- The only exception to this rule is the *java.lang* package, you can access its classes anywhere in your program without importing
 - Java makes it this way as it contains some very commonly used classes, such as: *System, Math, String, Integer, Float, Exception,*

Work with Java System Library

- Library: Collection of classes that can be used as basic building blocks of programs
- In Java, libraries are normally packed into modules and need to be included in the project *build path* for it to be used
- Java Base Library is mainly the collection of classes included in *java.base* module, and the Graphics Library in *java.desktop* module
 - Under each module are *packages* that contains different *classes*
 - In Eclipse, Java System Library is added to the build path automatically when a project is created



Using Java API

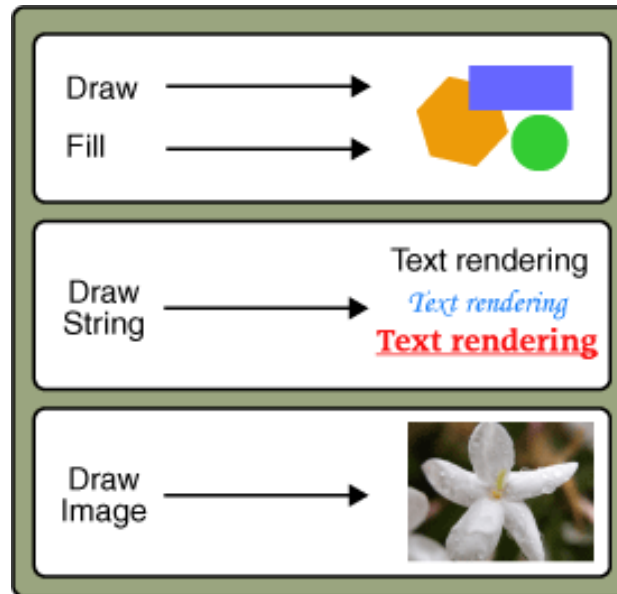
- **Application Programming Interface (API)** is a set of **publicly accessible** *methods, variables, and constants* of **classes** typically stored in a library
- Java API documentation is the primary reference for working with Java
 - <https://docs.oracle.com/en/java/javase/17/docs/api/index.html>
- Some commonly used **packages** in our context
 - java.lang – the only one NO need to *import*
 - **java.awt**
 - **javax.swing**

Intro to Java Graphics

- Java provides several graphics facilities that reflect the evolution of the language

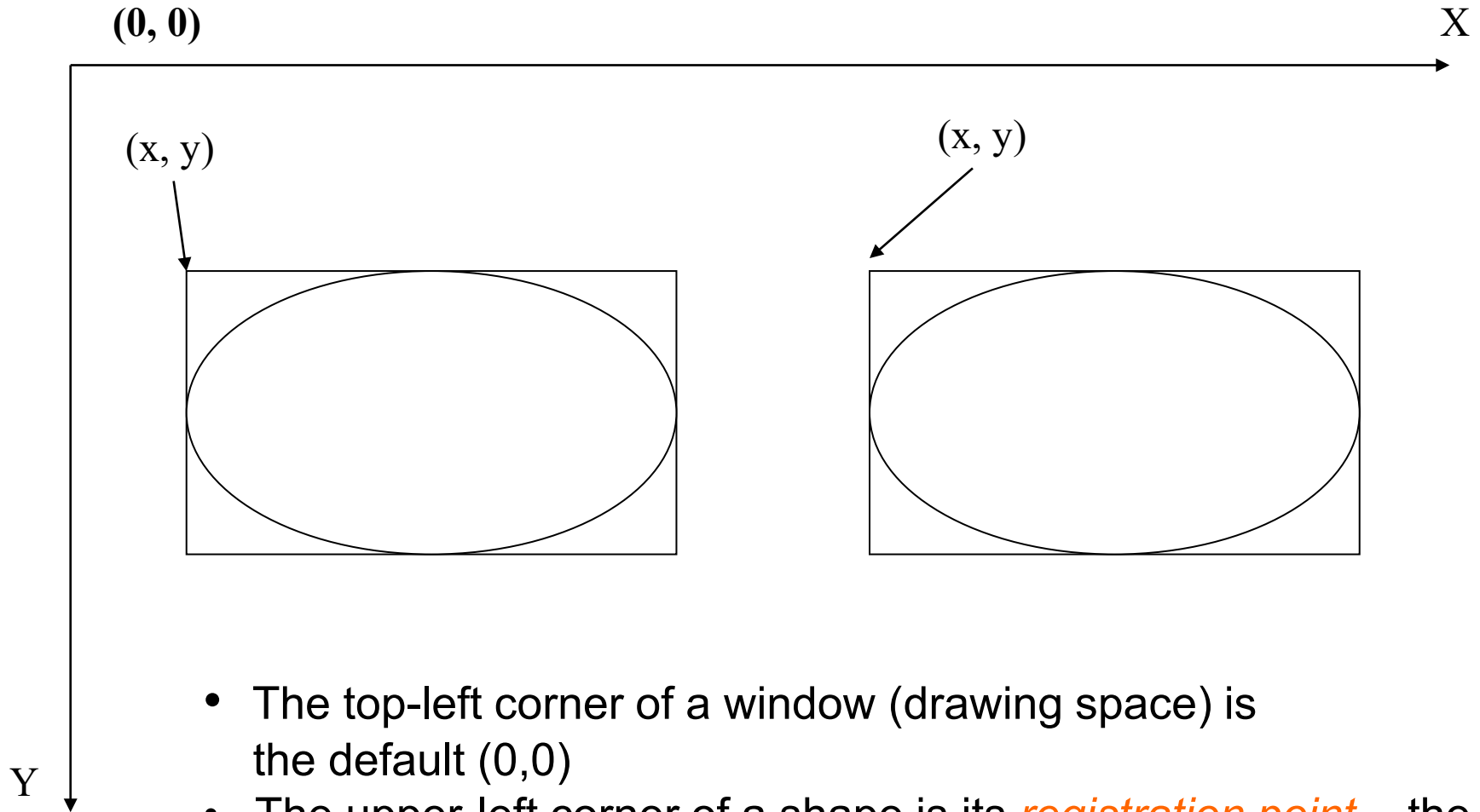
Methods of *Graphics* class

- Two basic groups of *Graphics*' methods:
 - *draw* and *fill* methods, enabling you to render basic shapes, text, and images



- Attributes *setting* methods, which affect how the drawing and filling appears
 - *setColor(Color color)*, *setFont(Font font)*

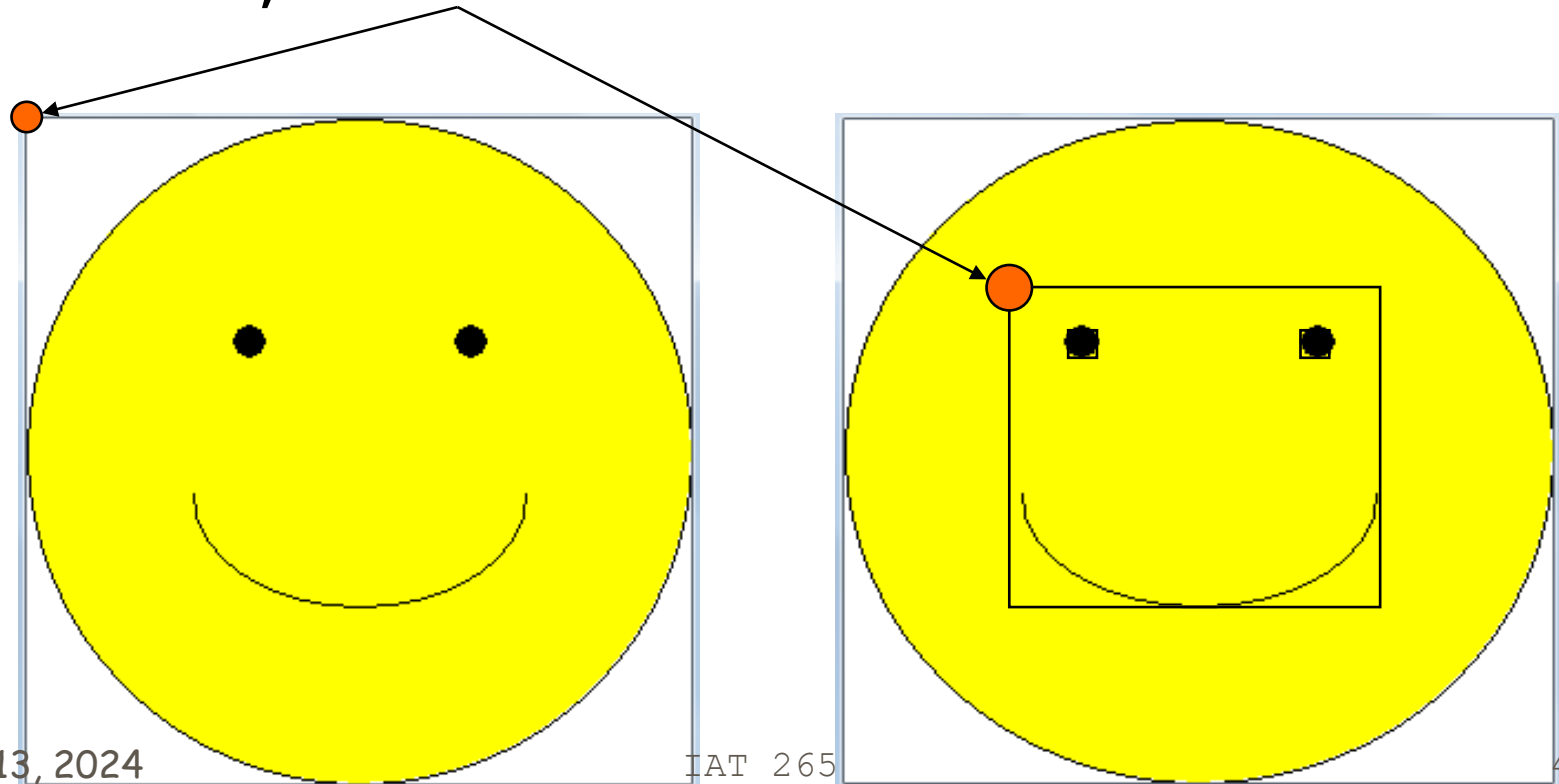
Drawing Space



- The top-left corner of a window (drawing space) is the default $(0,0)$
- The upper-left corner of a shape is its *registration point* - the starting point to draw a shape (including ellipse)

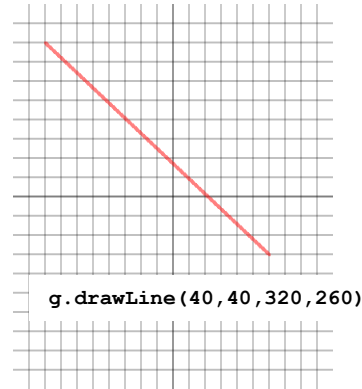
What is the **Registration Point** when drawing an Oval or Arc?

- It is the upper-left corner of the enclosing box, NOT that of the oval or arc

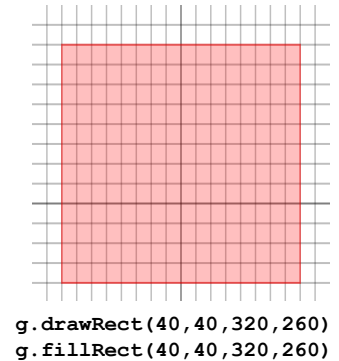


Drawing Primitive Shapes with *Graphics*' methods

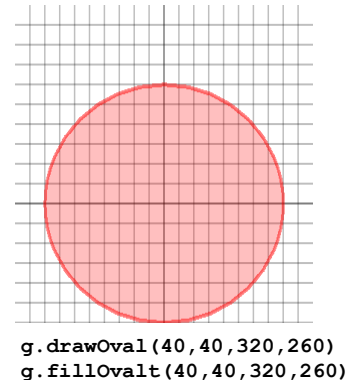
```
drawLine(int x1, int y1, int x2, int y2);
```



```
drawRect(int x, int y, int w, int h)  
fillRect(int x, int y, int w, int h)
```



```
drawOval(int x, int y, int w, int h)  
fillOval(int x, int y, int w, int h)
```

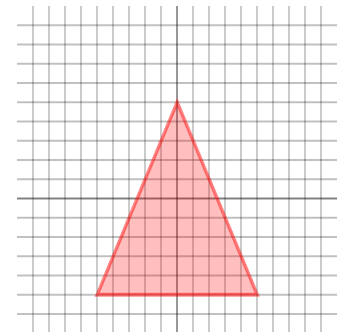


Drawing Primitive Shapes

```
drawPolygon(int xArray[], int yArray[], int numPoints)  
fillPolygon(int xArray[], int yArray[], int numPoints)
```

■ E.g. to draw & fill a triangle:

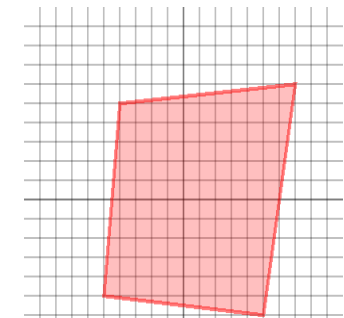
```
int[] xArray = {100, 200, 300};  
int[] yArray = {300, 100, 300};  
g.drawPolygon(xArray, yArray, 3);  
g.fillPolygon(xArray, yArray, 3);
```



```
g.drawPolygon(xArray,yArray,3)  
g.fillPolygon(xArray,yArray,3)
```

■ E.g. to draw & fill a quad:

```
int[] xArray = {100, 120, 340, 300};  
int[] yArray = {300, 100, 80, 320};  
g.drawPolygon(xArray, yArray, 4);  
g.fillPolygon(xArray, yArray, 4);
```



```
g.drawPolygon(xArray,yArray,4)  
g.fillPolygon(xArray,yArray,4)
```

Drawing arcs

`drawArc(int x, int y, int w, int h, int startAngle, int arcAngle)`

`fillArc(int x, int y, int w, int h, int startAngle, int arcAngle)`

`x, y` – the top-left corner of the arc's ellipse

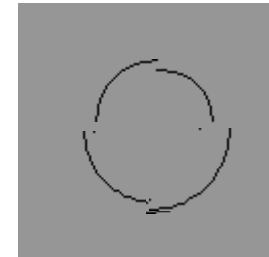
`w, h` – width, height of arc's ellipse

`startAngle` – angles to start the arc (in *degree*, positive for counter-clockwise)

`arcAngle` – the angular extent of the arc, relative to the start angle

■ Example:

```
g.drawArc(25, 25, 50, 50, 0, 90);  
g.drawArc(20, 20, 60, 60, 90, 90);  
g.drawArc(15, 15, 70, 70, 180, 90);  
g.drawArc(10, 10, 80, 80, 270, 90);
```



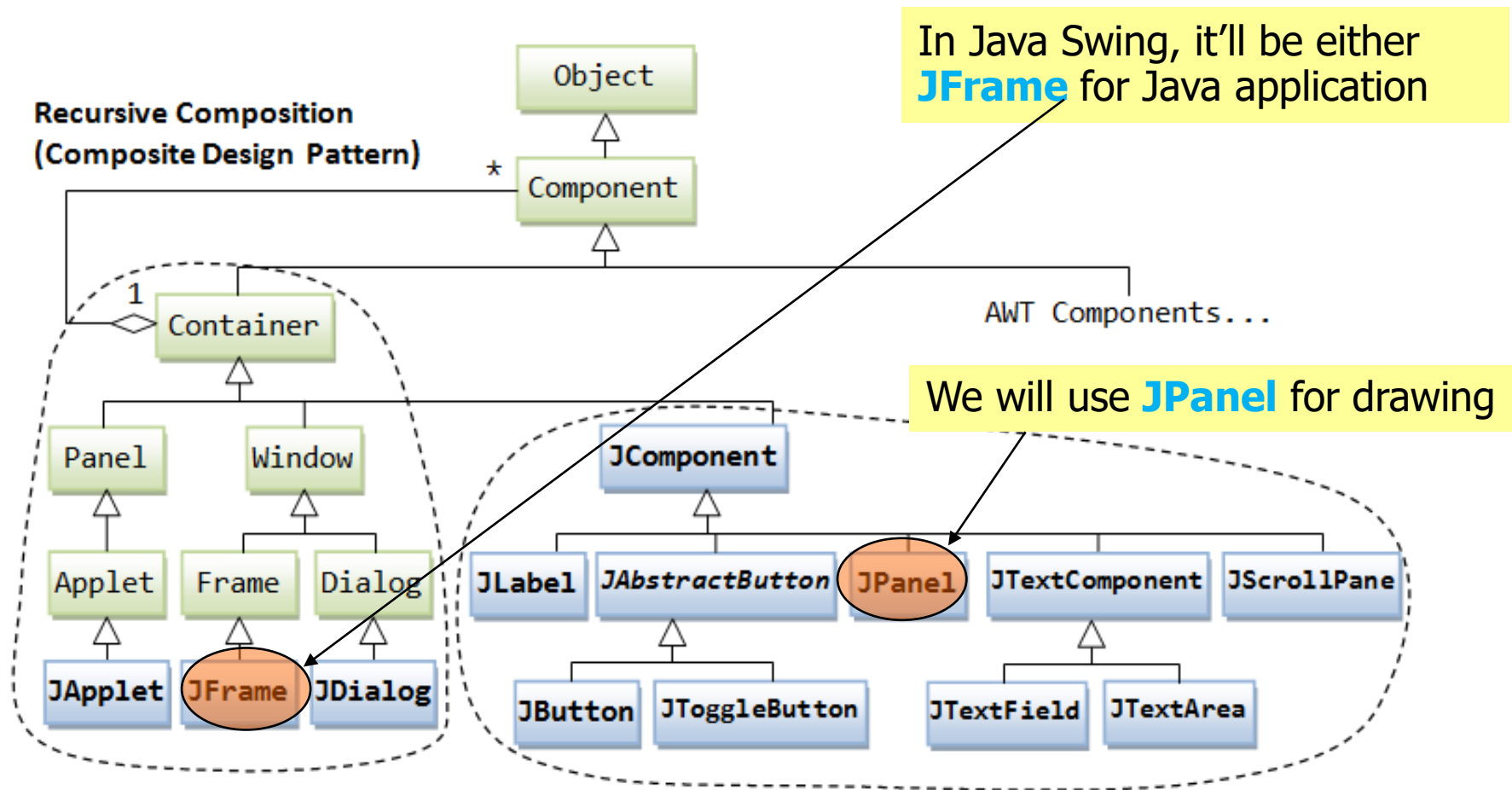
//or `g.drawArc(10, 10, 80, 80, -90, 90);` //for the last arc above

Where to draw?

- In Processing, the drawing space is implicit, you do your *setup()* and *draw()*
- In Java, you have to create and obtain the drawing surface explicitly

Inheritance Hierarchy of AWT & Swing Classes

- In order to see any graphics, we need to have a *top level container* for display and a panel for drawing



Metaphors for **JFrame** and **JPanel**



JPanel to **JFrame** is just like
the **sheet** to the **board**

- **Reference:** The image is from *KidKraft Deluxe Wooden Art Easel for Kids* at:
<http://www.brookstone.com/kidkraft-deluxe-wooden-art-easel-for-kids-w-paper-whiteboard-shelf>

First Drawing App

■ Create the window frame

```
public class BallApp extends JFrame {  
  
    public BallApp(String title) {  
        super(title);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //make closable  
        this.setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        new BallApp("BallApp");  
    }  
}
```


First Drawing App (1)

■ Create panel (canvas) for drawing

```
public class BallPanel extends JPanel {
    public final static int PAN_WIDTH = 600;
    public final static int PAN_HEIGHT = 450;

    public BallPanel() {
        super(); //call JPanel's default constructor
        this.setPreferredSize(new Dimension(PAN_WIDTH, PAN_HEIGHT));
    }

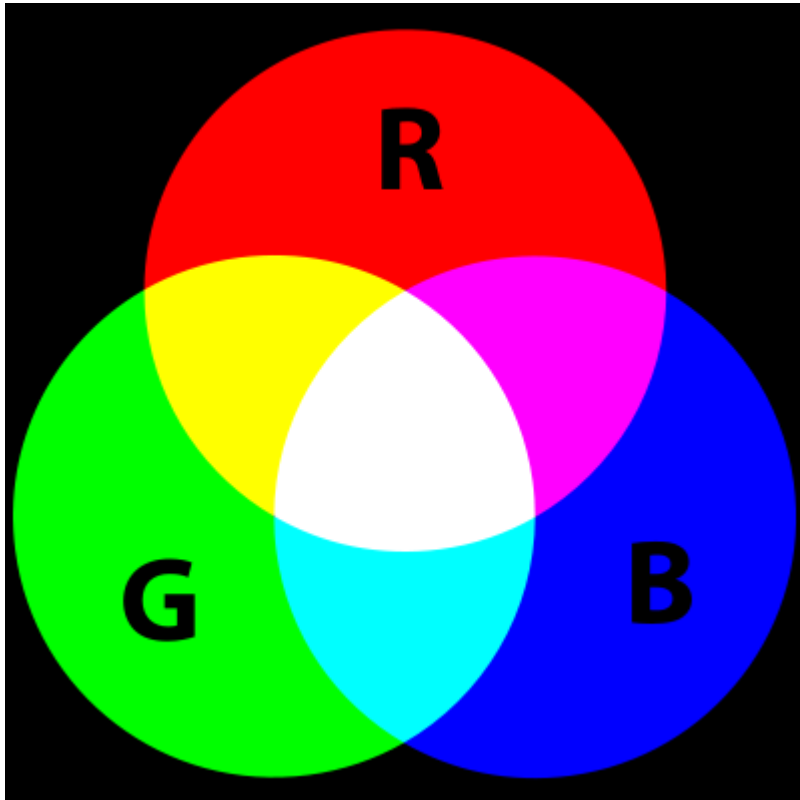
    public void paintComponent(Graphics g) {
        int ballW = 250;
        int ballH = 250;
        // make it center at the window's center
        int xPos = PAN_WIDTH/2-ballW/2;
        int yPos = PAN_HEIGHT/2-ballH/2;
        g.setColor(Color.RED);
        g.fillOval(xPos, yPos, ballW, ballH);
    }
}
```

First Drawing App (2)

- Add the panel to the window frame

```
public class BallApp extends JFrame {  
  
    public BallApp(String title) {  
        super(title);  
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        BallPanel ballPane = new BallPanel();  
        this.add(ballPane);  
        this.pack(); //window's size is determined by packing to BallPanel's size  
        this.setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        new BallApp("BallApp");  
    }  
}
```

RGB Color Model



- (255, 0, 0)
 - (160, 0, 0)
- (0, 255, 0)
- (0, 0, 255)
- (255, 255, 255)
- (0, 0, 0)
 - (150, 150, 150)

- Signature of Color class constructor:
Color(int r, int g, int b)

Color class has some **static Constants** for common colors

■ Including:

- *black* (or *BLACK*)
- *blue* (or *BLUE*)
- *cyan* (or *CYAN*)
- *green* (or *GREEN*)
- *orange* (or *ORGANGE*)
- *red* (or *RED*)
- *white* (or *WHITE*)
- *yellow* (or *YELLOW*)

■ As they are **static** member of *Color* class, you can access them using the class name, e.g:

- *Color.black*
- *Color.WHITE*
- *Color.yellow*

Review

■ Variables and Data Types

Variables

- A **variable** is a named memory location for storing a value
 - A variable has a *data type* associated with it, which determines the size of the storage location
- Things we do with variables:
 - Declaration `int x; int y;`
 - Initialization `x = 20; y = 40; //for the 1st time`
 - Assignment `x = 40; y = x; //post initialization`
 - Shortcut: `int x = 20; //merge declaration and initialization into one`

Why Data Types?

- Declaring data types tells the system (and you)
 - What kind of values to expect (error checking)
 - How to allocate space accordingly

- System uses types to detect errors

```
int pi = 3.14; //error:3.14 not an int
```



- To walk around, use *type casting* to enforce a variable from one type into another:

```
int pi = (int) 3.14 ;
```

```
print(pi) → 3
```



Review:

the “primitive” types

- byte – 8-bit signed integers between -128 and 127
- short – 16-bit signed integers between -32,768 and 32,767
- int – 32-bit signed integers between -2,147,483,648 and 2,147,483,647
- long – 64-bit signed integers between -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807
- float – 32-bit signed floating point numbers between $\pm 3.4E \pm 38$
- double – 64-bit signed floating point numbers between $\pm 1.7E \pm 308$
(default type for floating point numbers in Java)
- char – 16-bit, holds single character (e.g. 'c')
- boolean – 1-bit, holds the value either *true* or *false*

Classification of Data Types



■ Primitive Data Types:

- **integer** (byte, short, int, and long)
- **decimal** (float and double)
- **char** (E.g. a, b, c, A, B, C, &, *, etc)
- **boolean** (true or false)

■ Reference Data Types – object type:

- **Class, String, Array, ArrayList**

Difference between Primitive and Reference Types

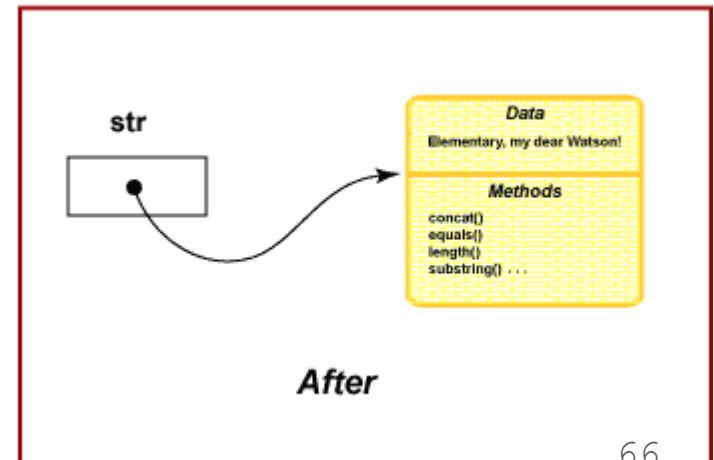
- A **primitive type** variable is an identifier for a value

- E.g. `int num = 10;`

num 10

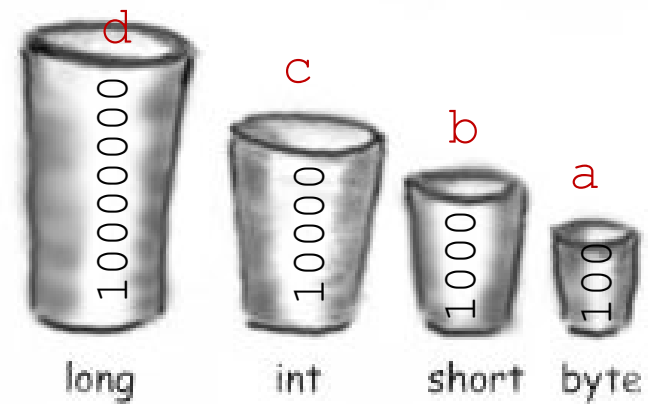
- A **reference type** variable is a reference to an object's memory location (its address rather than a value):

- E.g.
`String str = new String("Elementary, my dear Watson!");`



Another metaphor:

Primitive types



- A Primitive type variable is a bucket that holds values

byte: 8bits

e.g. **byte a = 100;**

short: 16bits

e.g. **short b = 1000;**

int: 32bits

e.g. **int c = 10000;**

long: 64bits

e.g. **long d = 100000000;**

Reference

- Like a remote control
- A **reference** is a primitive thing that **points at an object**
- The **assignment operator** links the reference to a **new** instance of the class

```
Dog d = new Dog();  
d.bark();
```



think of this
like this



¹
Dog myDog ³ = ² new Dog();

Reference Variable: Declaration & Initialization

- 1 Declare a reference variable

```
Dog myDog ;
```

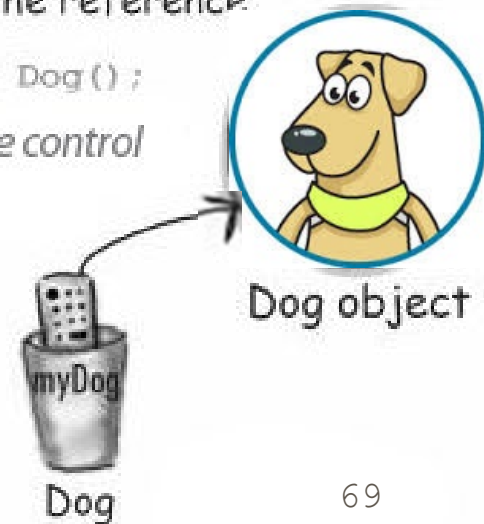


- 2 Create an object
new Dog();



- 3 Link the object and the reference.

```
Dog myDog = new Dog();  
programs the remote control
```



Summary

- Java: an OOP language
- Two types of Java program: Application and Applet
- File structure of Java program
- Method, parameter, argument
- Graphics object and its drawing primitives
- First drawing app

Readings

Required:

- Readings specified in Canvas

Optional:

- Applet versus Application