General Features of Java Programming Language

Variables and Data Types

Operators

Expressions

Control Flow Statements

Classes and Objects

Core Concepts in OOP

- Object
  - Model real-world objects as software objects
  - State of a program is composed of set of objects
- Class
  - A blueprint from which individual objects are created
  - It is a type

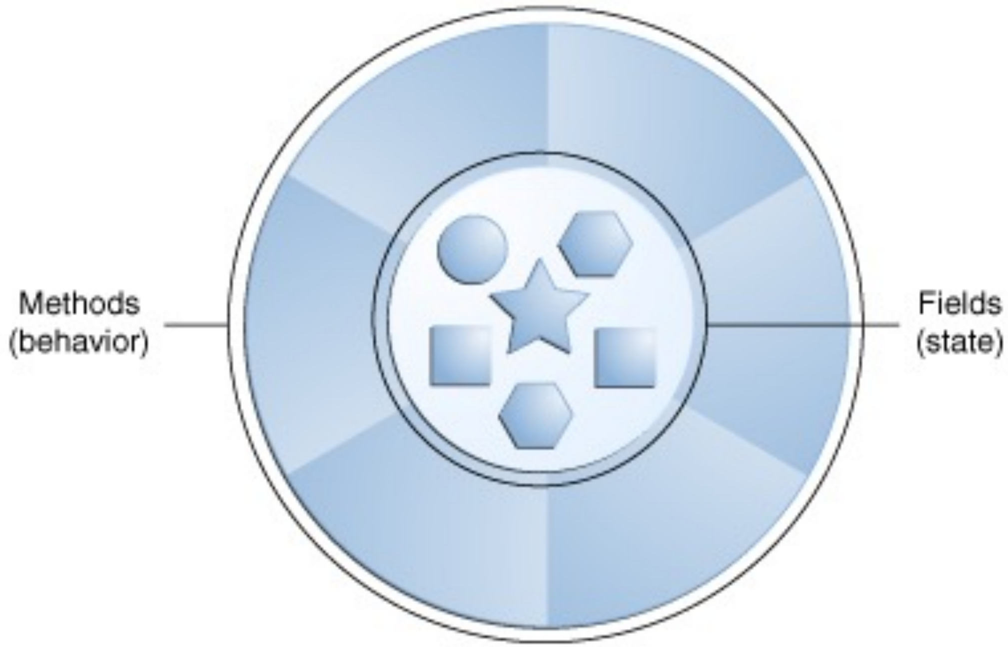
What is an Object?

- Real-world objects have state and behavior
  - dogs:
    - State: name, color, breed, hungry,…
    - Behavior: barking, fetching, wagging tail, …

- Software objects: state and behavior
  - Stores its state in *fields*
    - *State represents what an object knows*
  - Exposes its behavior through *methods*
    - Methods operate on an object's state
    - Serve as the primary mechanism for object-to-object communication

What is an Object - 2?

- Software Object



What is a Class?

- A class is the blueprint from which individual objects are created
- A *class* is a collection of
  - *fields/attributes* (data) and
  - *methods* (procedure or function) that operate on that data

```
class  ClassName [extends SuperClassName] {
        [fields declaration]

        [methods declaration]
}
```

State

Behavior

- A class is a type

Types in Java

- Two categories of data types in Java
  - primitive data type
    - Platform independent
  - reference data type:
    - class, interface, array

Data Type	Size
byte	8-bit
short	16-bit
int	32-bit
long	64-bit
float	32-bit
double	64-bit
char	16-bit Unicode
boolean	2 (false/true)

Fields/Attributes

- Two types: static and non-static
  - Syntax: [static] **type nameOfAttribute;**
- **Non-static** attributes
  - Specify the state of each instance (object) of the class
- Class Point:
  - A point **knows** two coordinates
    - **Attributes**: x and y

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

- Class Pen:
  - A pen **knows** its level and if it is closed or not
    - **Attributes**: level and closed

```
public class Pen {
  int level;
  boolean closed;
}
```

Accessing non-static Fields

- Use the dot (.) operator
  - ***reference***.fieldName
- *Outside context of object*
  - ***reference*** must be an object reference
- *Inside context of object*
  - *When processing a method invoked on the object*
  - ***reference*** *can be omitted*

```
Point p;
…
p.x = 2;
p.y = 4;
..
```

```
public class Point {
  …
  public int getX() {
    return x;
  }
}
```

Methods

- Two types of methods
  - static and non-static
- Each non-static method
  - is **always invoked** on the context of an object
  - has access to
    - all local variables of the method
    - parameters of the method
    - and state of the invoked object (non-static attributes)
    - and static attributes
- Method Invocation
  - similar to field access
  - use the dot (.) operator
    - `reference.method(arguments)`
    - "`reference`" must be an object reference
    - `reference can be omitted`

Example: Point Class

Example: Point class:

- State
    - x and y
- Functionality
    1. Get value of each attribute
    2. Change each attribute
    3. Move point given *(dx, dy)*

<u>Point.java</u>

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

  int getX() {
    return x;
  }

  void setX(int newX) {
    x = newX;
  }

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

   // remaining methods for y
}
```

Creating Objects

- Apply **new** special operator
  - It is a keyword
  - Syntax: *new ClassName()*
    - Returns an instance of ClassName
  - See more detail later

Invoking Methods

```
public class Main{
  public static void main(String[] args) {
    Point p1 = new Point();
    Point p2 = new Point();
    // state of p1?          (0, 0)
    p1.setX(2);
    p1.setY(3);
    // state of p1?          (2, 3)
    p1.move(2, 2);
    // state of p1?          (4, 5)
    p2.setX(4);
    p2.setY(6);
    p2.move(2, 2);

    // state of p1?          (4, 5)

    // state of p2?          (6, 8)
  }
}
```

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

  int getX() {
    return x;
  }

  void setX(int newX) {
    x = newX;
  }

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

   //…
```

New requirement for Point

- Know the number of created points

- How to keep this information?

- Add a new attribute to the class (numberOfPoints)?

Example: Point Class

Point.java

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

  int getX() {
    return x;
  }

  int getY() {
    return y;
  }

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

Main.java

```
public class Main{
   public static void main(String[] args) {
      Point p1 = new Point();
      p1.numberOfPoints++;
      p1.x = p1.y = 5;
      Point p2 = new Point();
      p2.numberOfPoints++;
      p2.x = p2.y = 6;

      // value of p2?  (6, 6)  p1?   (5, 5)
      p2.move(2, 3);
      // value of p2?  (8, 9)  p1?   (5, 5)
      // value of p1.numberOfPoints  1
      // value of p2.numberOfPoints  1
   }
}
```

- Having *numberOfPoints* as a (non-static) attribute of Point **does not** solve the problem!

Static fields

- Set of non-static fields specify the state of each instance of the class
- Set of static fields of a class
  – Specify the *state* of the class
  – Static fields are **shared** by all instances of a class
  – Declare attribute using the **static** keyword

- How to store the number of created points?
  – Should use a static field in Point

Accessing static Fields

- Use the dot (.) operator
  - ***reference***.fieldName

- Static field
  - Inside context of class
    - ***reference*** *can be omitted*
  - Outside context of class
    - ***reference*** = ClassName
    - ***reference*** = reference to an instance of the class

Example: Point Class

Point.java

```
public class Point {
  int x;
  int y;
  static int numberOfPoints;

  int getX() {
    return x;
  }

  int getY() {
    return y;
  }

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

Main.java

```
public class Main{
  public static void main(String[] args) {
    Point p1 = new Point();
    p1.numberOfPoints++;
    p1.x = p1.y = 5;
    Point p2 = new Point();
    p2.numberOfPoints++;
    p2.x = p2.y = 6;

    // value of p2?  (6, 6)  p1?  (5, 5)
    p2.move(2, 3);
    // value of p2?  (8, 9)  p1?  (5, 5)
    // value of p1.numberOfPoints  2
    // value of p2.numberOfPoints  2
    // value of Point.numberOfPoints  2
  }
}
```

- Having *numberOfPoints* as a static attribute of Point **solves** the problem!

Invocation of static Methods

- Invoked as operations on classes using the dot (.) operator

```
reference.method(arguments)
```

- Outside of the class: “`reference`” can either be the class name or an object reference belonging to the class
- Inside the class: “`reference`” can be omitted
- Can access to
  - parameters of the method
  - local variables
  - and **only static** atributtes
- Always invoked on the context of a class

Example: Point Class

Define a method that returns number of created points

Point.java

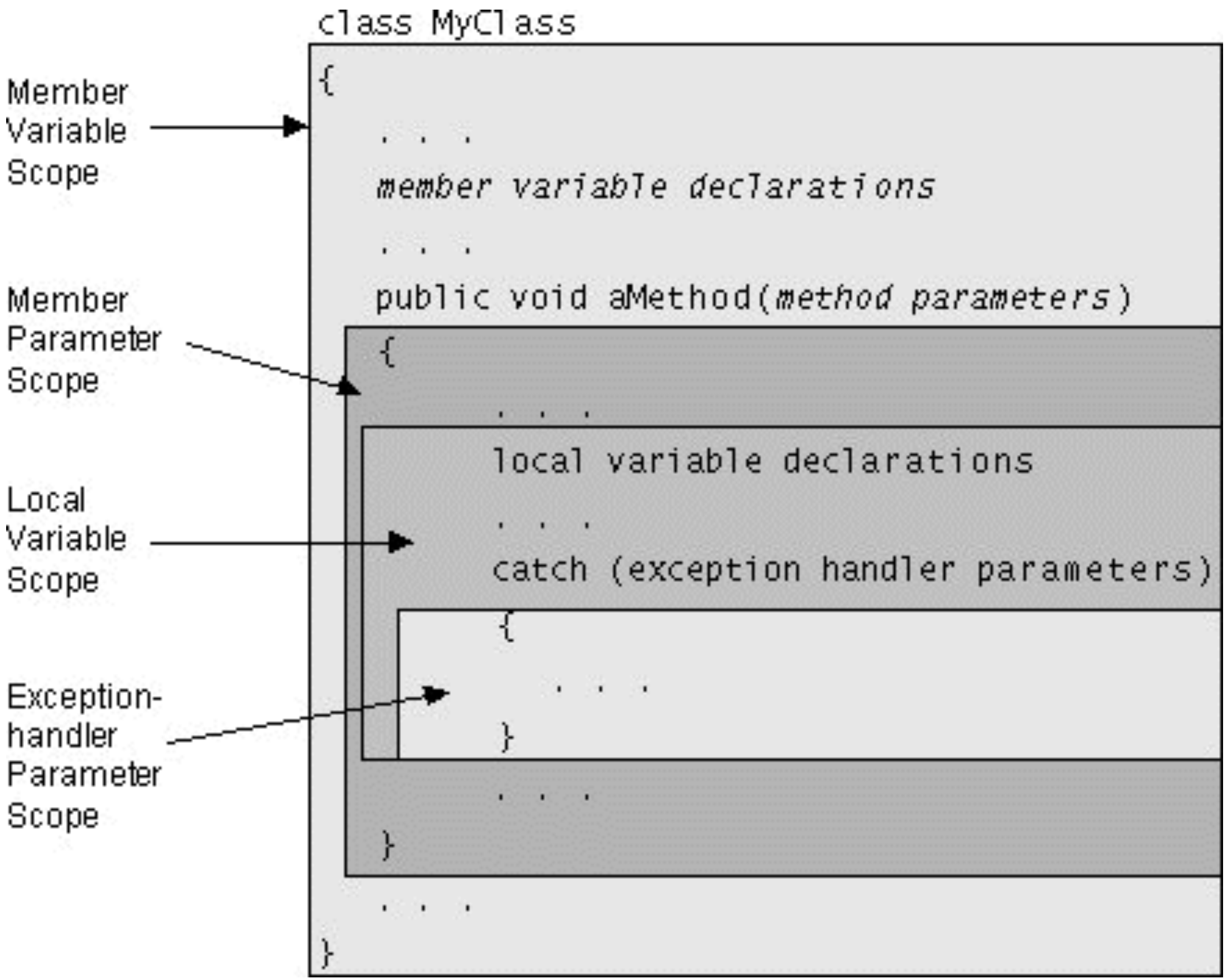
```
public class Point {

  static int getNumberOfCreatedPoints() {
    return numberOfCreatedPoints;
  }

  public static void main(String[] args) {
    Point p1 = new Point();
    Point.getNumberOfCreatedPoints();
    p1.getNumberOfCreatedPoints();
    getNumberOfCreatedPoints();
}
```

Which invocations are correct?
All

Variable Scope

- The block of code within which the variable is accessible and determines when the variable is created and destroyed.
- The location of the variable declaration within your program establishes its scope
- Variable Scope:
  – Member variable
  – Local variable
  – Method parameter
  – Exception-handler parameter

Variable Scope



Operators

- Operators perform some function on operands
- An operator also returns a value

Operators (I)

- Arithmetic Operators
  - Binary: +, -, *, /, %
  - Unary: +, -, op++, ++op, op--, --op
- Relational Operators
  - >, >=, <, <=, ==, !=   (return *true* and *false*)
- Conditional Operators
  - &&(AND), ||(OR), !(NOT), &(AND), |(OR)
  - expression ? op1 : op2
- Bitwise Operators
  - >>, <<, **>>>,** &, |,^,~

```
(i>5) ? j=1 : j=2
```

```
if (i>5)
  j=1;
else
  j=2;
```

Operators (II)

- Assignment Operators
  - =
  - +=
  - -=, *=, /=, %=, &=, !=, ^=, <<=, >>=, >>>=

$$\boxed{op1 \mathrel{+}= op2} \Longrightarrow \boxed{op1 = op1 + op2}$$

$$\boxed{s \mathrel{+}= 2} \Longrightarrow \boxed{s = s + 2}$$

Expressions

- Perform the work of a Java Program

- Perform the computation

- Return the result of the computation

Expression

- An expression is a construct made up of variables, operators, and method invocations
  - respects the syntax of the language
  - evaluates to a single value
    - count++
    - System.out.println(”Value at index 1 ” + array[1]);
- Precedence
  - Precedence Table
  - Use (.....)
- Equal precedence
  - Assignment: Right to Left (a = b =c)
  - Other Binary Operators: Left to Right

Statement

- Statements are roughly equivalent to sentences in natural languages
- A *statement* forms a complete unit of execution.
- *Statement*: an expression terminated with a semicolon (;)
  - Assignment statement: aVar = 5;
  - Method invocation: System.out.println(“Hello!”);
  - Declaration statement: int aVar;
  - …

Blocks

- A *block* is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.

```
class BlockDemo {
  public static void main(String[] args) {
    int i = 0;
    boolean condition = someInvocation();
    if (condition) { // begin block 1
      System.out.println("Condition is true.");
      i++;
    } // end block 1
    else { // begin block 2
      System.out.println("Condition is false.");
    } // end block 2 }
}
```

Control Flow Statement

If Statements

- if (*boolean*) {
  /* ... */
  }
  else if (*boolean*) {
  /* ... */
  } else {
  /* ... */
  }

Statement Block

- The expression in the test must return a *boolean* value
  - In Java, zero(‘**0**') cannot be used to mean false, or non-zero("...") to mean true

Example

```
if (income < 20000) {
  System.out.println (“poor”);
}
else if (income < 40000) {
  System.out.println (“not so poor”);
}
else if (income < 60000) {
  System.out.println (“rich”);
}
else {
  System.out.println (“ very rich”);
}
```

Loops

- Three types of loops:

```
while (boolean expression) {
  /* ... */
}
```

```
do {
  /* ... */
} while (boolean expression)
```

```
for (expression; boolean expression; expression) {
  /* ... */
}
```

The **for** statement

Reserved word

The initialisation part is executed once before the loop begins

The statement is executed until the condition becomes false

for (initialisation; condition; increment)
statement;

The increment part is executed at the end of each iteration

Example

Count from 1 to 10 and write value

while loop

```
int i = 1;
while (i <= 10) {
        System.out.println (i);
        i++;
}
```

do-while loop

```
int i = 1;
do {
        System.out.println (i);
        i ++;
} while (i < 10);
```

for loop

```
for (int i = 1; i <= 10; i++) {
        System.out.println (i);
}
```

Secret! A `for` loop can almost always be converted to a `while` loop

```
for(i = 0; i < 10; i++) {
   body of loop;
}
```

```
i = 0;
while (i < 10) {
   body of loop;
   i++;
}
```

- *This will help you understand the sequence of operations of a* **`for`** *loop*
- *What is the exception?*

Control Flow: Branching break (unlabeled)

```
void search(int[][] array, int searchFor) {
  boolean foundIt = false;

  for (int i = 0; i < array.length; i++) {
    for (int j = 0; j < array[i].length; j++) {
      if (array[i][j] == searchfor) {
        foundIt = true;
        break;
      }
    }
    if (foundIt)
      break;
  }
  if (foundIt) {
    System.out.printf("Found %d at %d,%d", searchfor, i, j);
  } else {
     System.out.printf(" %d not in the array", searchfor);
  }
}
```

Control Flow: Branching break (labeled)

```
void search(int[][] array, int searchFor) {
  boolean foundIt = false;

search:
  for (int i = 0; i < array.length; i++) {
    for (int j = 0; j < array[i].length; j++) {
      if (array[i][j] == searchfor) {
        foundIt = true;
        break search;
      }
    }
  }
  if (foundIt) {
    System.out.printf("Found %d at %d,%d”, searchfor, i, j);
  } else {
     System.out.printf(“ %d not in the array”, searchfor);
  }
}
```

Strings in Java

- A string in Java is represented by an instance of a special class
  - String

- Creating a string:

  - String ex2 = new String("abc");

  - String ex = "abc";

Switch Expression

```
switch (expression) {
  case constante1:
     //…
     break;  // is optional
  case constante2:
     //…
     break;  // is optional
  …
  default:  /* is optional */
     /* … */
     break;  // Needed???
}
```

- Expression can be of primitive type, String, enumerate
- For safety and good programming practice, ***always*** include a 'default' case.
  - With a **break** statement

Integer Literals

- **Integer Literals**
  - Integer literal is **long** if it ends with L or l
  - Otherwise is **int**
- Can be expressed by
  - Decimal: Base10
    - digits 0 through 9
  - Hexadecimal: Base 16 (leading 0x)
    - digits 0 through 9 and letters A through F
    - **0x**AB23 <u>*(0x1f = 31)*</u>
  - Binary: Base 2 (leading 0b)
    - digits 0 and 1
    - **0b**1001
  - Octal: Base 8 (leading 0)
    - digits 0 through 7
    - **011**

Floating Point Literals

- Floating point literal is **float** if it ends with F or f
  - 3.1415f *(32-bit Float)*
- Otherwise is double
  - Optionally end with the letter D or d.
  - 6.1D *(64-bit Double; Default)*
  - *6.1* *(64-bit Double; Default)*

Underscore Characters in Numeric Literals

- Underscore characters (_) can appear anywhere between digits in a numerical literal
- Improves readability of your code
- Examples:
  - long creditCardNumber = 1234_5678_9012_3456L
  - long bytes = 0b1101000_0110101_1001010_1001001;
  - float pi = 3.14_15F;

Character and String Literals

- Characters
  - '.....' e.g. 'a'
  - '\t', '\n'   (Escape Sequence)
- Strings
  - "....... " e.g. "Hello World!"
  - String Class (Not based on a primitive data type)

Reserved Words

(Keywords)

`abstract`	`default`	`if`	`private`	`throw`
`boolean`	`do`	`implements`	`protected`	`throws`
`break`	`double`	`import`	`public`	`transient`
`byte`	`else`	`instanceof`	`return`	`try`
`case`	`extends`	`int`	`short`	`void`
`catch`	`final`	`interface`	`static`	`volatile`
`char`	`finally`	`long`	`super`	`while`
`class`	`float`	`native`	`switch`	
`const`	`for`	`new`	`synchronized`	
`continue`	`goto`	`package`	`this`	

Don't worry about what all these words mean or do, but be aware that you cannot use them for other purposes like variable names.

Variables

- Java supports 4 types of variables:
  - **Instance Variables (Non-Static Fields)**
    - their values are unique to each *instance* of a class
  - **Class Variables (Static Fields)**
    - Their values are unique to each class
  - **Local Variables**
    - Methods store their temporary state in *local variables*
  - **Parameters**

Variables - 2

- Java is statically-typed
  - Every variable must first be declared before it can be used
  - Every variable must have a type
  - Declaration syntax:
    - Type verName [= initialValue];
- A variable's data type determines its value and operation
- Two categories of data types in Java
  - primitive data type: byte, short, int, long, float....
  - reference data type: class, interface, array

Variable Initialization

- Initial value is not mandatory
- Default value: behavior depends on kind of variable
- For fields (static and non-static)
  - There is a default value (depends on variable type)
    - 0 for primitive numerical type
    - false for boolean
    - ***null*** for reference type
- For local variables
  - Compiler never assigns a default value to an uninitialized local variable
  - Accessing an uninitialized local variable will result in a compile-time error

Variable Names

- Java refers to a variable's value by its name
- General Rule : must be a legal Java identifier
  - **[A-Z.a-z.$,_][[A-Z,a-z,,0-9,$,_])***
  - Must not be a keyword or a boolean literal
  - Must not be the same name as another variable in the same scope

- Convention in this course (**mandatory**):
  - Name must be related to the function of the variable/method/class
  - Variable names begin with a lowercase letter
    - Examples: empty, visible, count, input
    - Fields should start with a '_'
    - Starts with a noun
    - If more than one word, remaining words are uppercased
  - Class names begin with an uppercase letter
    - Starts with a noun
    - If more than one word, remaining words are uppercased
      - ArrayList

Convention for Method Names

- Method Names
  - Begin with an lowercase letter
  - Starts with a verb
  - If more than one word, remaining words are uppercased
  - Examples: bark(), waggleTail()

Example: Point Class

Point.java

```
public class Point {
  int x;
  int y;
  static int numberOfPoints;

  int getX() {
    return x;
  }

  int getY() {
    return y;
  }

  void setX(int nx) {
    if (nx > 0)
      x = nx;
  }

  void setY(int ny) {
    if (ny > 0)
      y = ny;
  }
}
```

Main.java

```
public class Main {
  public static void main(String[] args) {
    Point p1 = new Point();
    p1.x = -5;
    Point.numberOfPoints = -4;
  }
}
```

- Suppose that Point class has constraint:
  - Each Point must have positive x and y
  - What do I need to change?
- Problems of proposed solution?

1. Does not ensure a consistent state of object/class
2. Can create objects with a inconsistent state
3. If internal representation of the state changes it has impact on the clients (more general)

Encapsulation

- A fundamental concept of OO programming languages
- Hide sensitive information and/or implementation detail
- Benefits of Encapsulation
  - A class can have total control over what is stored in its fields
  - A class can define methods that can only be used by the class itself
  - A class can change its implementation without an impact on its clients
- Java implements this mechanism with access level modifiers

Controlling Access to Members of a Class

- Member = Field or Method
- Several access level modifiers
  - determine whether other classes can use a particular field or invoke a particular method
  - Fundamental for ensuring information hiding

- There are two levels of access control
  - Top level (or class level)
    - public or *package-private* (no explicit modifier)
  - At the member level
    - public, private, protected, or *package-private* (no explicit modifier).

Top Level Access Control Modifiers

- *public*
  - Class is visible to all classes everywhere
- *package*
  - Class is visible only within its own package
  - Package is a named group of related classes
    - More later

Member Level Access Control Modifiers

- *private:* only in the class itself
- *package:* classes in the same package and the class itself
- *protected:* classes in the same package, subclasses of the class, and the class itself
- *public:* anywhere the class is accessible

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Tips on choosing Access Level

- Goals:
  - Hide implementation details
  - Ensure that errors from misuse cannot happen

- Called Encapsulation/Information Hiding

- *Solution:*
  - Use the most restrictive access level that makes sense for a particular member
    - By default use ***private*** access level
  - **Avoid** public fields
    - Except for constants

A better Point Class

Point.java

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

   private static int numberOfPoints

   public int getX() {
        return x;
   }

   public int getY() {
        return y;
   }

    public void move(int dx, int dy) {
        x += dx;
        y += dy;
   }
   public static void incNumberOfPoints() {
        numberOfPoints++;
   }
   public static int getNumberOfPoints() {
        return numberOfPoints;
   }
}
```

Main.java

```
public class Main{
   public static void main(String[] args) {
       int x = Point.numberOfPoints;

       Point p1 = new Point();
       Point p2 = p1;
       Point.numberOfPoints++;

       x = p1.x;;
       p2 = new Point();
       p2.numberOfPoints++;
   }
}
```

Creating Objects

- Objects are created using the **new** operator
- Each object should be created in a valid state
- The initialization of the each new object belongs to the responsibility of the class
  - Need to have initialization code

- How to ensure object is created in consistent state?

Initialization code

Point.java

```
public class Point {
  private int _x;
  private int _y;

  private static int _numberOfPoints

  public int getX() {
    return _x;
  }

  public int getY() {
    return _y;
  }

  public void init(int x, int y) {
    if (x > 0)
      _x = x;
    if (y > 0)
      _y = y;
  }
  // …
}
```

- How to ensure consistent state after creation?
- Add a method responsible for this
- Problem of proposed solution?

```
public class Main {
  public static void main(String[] args) {
    Point p1 = new Point();
    p1.init(2, 3);
    Point p2 = new Point();
    p2.init(2, -3);

    p2.getY(); // value of y?  0
    // Does it respects restriction?
  }
}
```

Creating Objects - Solution

- Initialization code placed in special method called constructor
- It is automatically executed when an instance is created
- Name of constructor is equal to the name of the class
- Have **NO** return type (it is not **void**)
- Can have any access modifiers

Constructor

- Constructors with no arguments are called *no-arg* constructors
- Java ensures that all classes have at least one constructor
- If programmer does not specify any constructor
  - Java provides a *default no-arg* constructor
    - It does nothing
    - It has the same accessibility as its class

- A class can have more than one constructor with the same name as long as they have different parameter list
  - *Called overloading*
  - Each one represents a distinct way of initializing a new instance

Constructor - Example

Point.java

```
public class Point {
  private int _x;
  private int _y;

  private static int numberOfPoints

  public Point(int nx, int ny) {
    _x = nx;
    _y = ny;
  }

  public Point(int nx) {
    _x = nx;
    _y = 0;
  }

  // …
}
```

- We are considering in this example a class Point without any restriction

The this() Constructor

- When we have several alternative constructors, we need to have a way of reuse constructor methods
  - Otherwise we may have code duplication
- Java allows calling one constructor from another one
  - Use *this(parameters)* to invoke the desired constructor
  - The compiler determines which constructor to call, based on the number and the type of arguments

```
public Point(int nx) {
    this(nx, 0);
    // could have more code here
}
```

  - The invocation of another constructor **must** be the first line in the constructor

Example

Point.java

```java
public class Point {
  private int _x;
  private int _y;

  private static int _numberOfPoints

  public Point(int nx, int ny) {
    _x = nx;
    _y = ny;
  }

  public Point(int nx) {
    this(nx, 0);
  }

  // …

  public static void incNumberOfPoints() {
    _numberOfPoints++;
  }
  public static int getNumberOfPoints() {
    return _numberOfPoints;
  }
}
```

Main.java

```java
public class Main {
  public static void main(String[] args) {
    Point p1 = new Point(1, 3);
    Point.incNumberOfPoints();
    Point p2 = new Point(1);
    p2.incNumberOfPoints();
    // …
  }
}
```

- What is the problem with this code?
- Point class does not ensures that *_numberOfPoints* is correct
- This depends on client code!
  - Never do this
- How to have it right?

Example - 2

Point.java

```
public class Point {
  private int _x;
  private int _y;

  private static int _numberOfPoints

  public Point(int nx, int ny) {
    _x = nx;
    _y = ny;
    _numberOfPoints++;
  }

  public Point(int nx) {
    this(nx, 0);
  }

  // …

  public static int getNumberOfPoints() {
    return _numberOfPoints;
  }
}
```

Main.java

```
public class Main{
  public static void main(String[] args) {
    Point p1 = new Point(1, 3);
    // Point.incNumberOfPoints();
    Point p2 = new Point(1);
    // p2.incNumberOfPoints();
    // …
  }
}
```

- Everything right?
- Need to add “numberOfPints++” to ctor Point(int)?
- NO!

Creating Object

- There are three steps when creating an object from a class: **Declaration**, **Instantiation and Initialization**
  - ClassName varName = **new** ClassName();
- **Declaration**
  - Left part declares a variable with name *varName*
  - Variable holds a **reference** to an object
- **Instantiation**
  - **new** operator
    - Allocates memory in heap for new object
    - Returns reference for that memory
- **Initialization**
  - Initializes non-static fields with default values
  - Execute constructor

Initialization Order

- Java ensures that code cannot access uninitialized fields
- **Static fields** are initialized when the class is referred first time in execution:
  1. Initialize static fields to default values
  2. Initialize static fields using value (if present) in declaration
  3. Execute static initialization blocks

- **Non-static fields** are initialized every time an object is created:
  1. Initialize non-static fields to default values
  2. Initialize non-static fields using value (if present) in declaration
  3. Execute initialization blocks
  4. Execute constructor

Initialization of Fields

- There are several ways for initializing fields

1. Use default value
   – ***null*** represent the null reference

Data Type	Default Value
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false
Object reference	null

Initialization of Fields - 2

2. Provide an initial value for a field in its declaration

```
public class BedAndBreakfast {
    // initialize to 10
    public static int capacity = 10;
    // initialize to false
    private boolean full = false;
}
```

- Simple
- Not very powerful

Initialization of Fields - 3

3. Static Initialization Blocks
   – A normal block of code enclosed in braces, { }, and preceded by the **static** keyword
     • Usually, used to initialize the static fields of the class

```
public class Whatever {
    static {
      // whatever code is needed for initialization goes here
    }
}
```

     • Can have several static initialization blocks
       – Anywhere in class body
       – Executed by the same order
   – Executed once
   – There is an alternative
     • Write a private static method
     • Can reuse code

```
class Whatever {
   private static varType myVar = initializeClassVariable();

   private static varType initializeClassVariable() {
      // initialization code goes here
   }
}
```

Initialization of Fields - 4

4. Initialization Blocks
   - A normal block of code enclosed in braces, {}

```
public class Whatever {
   {
     // whatever code is needed for initialization goes here
   }
}
```

   - Usually, used to initialize non-static fields of the class
   - Can have several initialization blocks
     - Anywhere in class body
     - Executed by the same order
   - Executed when an object is created
   - There is an alternative
     - Write a protected final method
     - Can reuse code

```
class Whatever {
   private varType myVar = initializeInstanceVariable();

   protected final varType initializeInstanceVariable() {
      // initialization code goes here
   }
}
```

Initialization of Fields - 5

5. Constructors
   – More powerfull

Final Fields

- A field/variable can be *constant*
  - **Cannot** change value of a *final* field after it has been initialized
  - Use *final* keyword
- Initialization of *final* fields
  - Static final fields
    - In declaration
    - In static initialization blocks
    - Can be **public**
  - Non-static final fields
    - In declaration
    - In initialization blocks
    - In constructors
- A final non-static field is distinct from a final static field
- Other types of variable (parameters and local) can also be final
- You can view final variables as constants
  - Variable always hold the same value
  - But, if variable holds a reference …

Initialization Order - Example

```
public class ExecutionDemo {
  public static void main(String[] args) {
    System.out.println("Begin ExecutionDemo");
    DemonstrateOrder demo = new DemonstrateOrder();
    DemonstrateOrder demo2 = new DemonstrateOrder();
  }
}
class DemonstrateOrder {
  private static String vstatic = p1("Static declaration field initialization”, “declaration”);
  private String vnonstatic = p2("Declaration field initialization”, “declaration”);

  static { vstatic = p1("Static initialization block”, “block”); }

  { vnonstatic = p2("Initialization block”, “block”); }

  static String p1(String msg, String value) {
    System.out.println("In " + msg + ”: vstatic  -> " + vstatic);
    return value;
  }
  protected final String p2(String msg, String value) {
    System.out.println("In " + msg + ”: vnonstatic -> " +
                       vnonstatic);
    return value;
  }
  public DemonstrateOrder ( )  { p2("In constructor”, “ctor”); }
}
```

Result when execute ExecutionDemo?

```
Begin ExecutionDemo
In Static declaration field initialization: vstatic -> null
In Static initialization block: vstatic -> declaration
In Declaration field initialization: vnonstatic -> null
In Initialization block  vnonstatic: -> declaration
In constructor vnonstatic: -> block
In Declaration field initialization: vnonstatic -> null
In Initialization block: vnonstatic -> declaration
In In constructor: vnonstatic -> block
```

Wrapper Classes

- There is a class version for each primitive type

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	Boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float or String
double	Double	float, double or String
long	Long	long or String
short	Short	short or String

Autoboxing and Unboxing

- Java implicitly boxes primitive types into the appropriate wrapper class when necessary
- And also unboxes the primitive type from the wrapper class
  - char to/from Character, int to/from Integer, double to/from Double, etc...
- Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably

```
Long refLong = 2;
refLong = 3L + refLong;
```

Concatenation Operator

String str = “abc” **+** “def”;

- Operator + is special when left operand is a String
- In this case, + is concatenation
  - Returns a reference to a String whose content is the concatenation of the first String with the string that represents the second operand
    - Creates a new String instance
    - str holds a reference to “abcdef”
  - If 2nd operand is not String
    - Primitive value -> convert into the corresponding string
      - “abc” + 1 -> “abc1”
    - A reference -> invoke toString()

Life Cycle of dynamically created Objects

- C
  - malloc() – use – free()
- C++
  - new() – constructor() – use – destructor()
- Java
  - new() – constructor() – use – [ignore / garbage collection]
  - Dynamic memory is automatically managed by a special entity: **Garbage Collector**
    - Reclaim space from other no longer used objects
    - An object is unused if the program holds no more references to it.
    - How to drop a reference?
      - Set the variable holding the reference to *null*

finalize() Method

- Garbage collection ONLY frees the ***memory resources***
- How to free other types of resources?
  - network connection, DB connection, file handler
- Java ensures that *finalize()* is invoked before destroying the object
- Specify the actions to release the resources in
  - protected void finalize() throws Throwable
- How to properly write a finalize() method?

```
protected void finalize() throws Throwable {
    try {
        // clean up
    } finally {
        super.finalize(); // call parent class' finalize
    }
}
```

finalize() Method Drawback

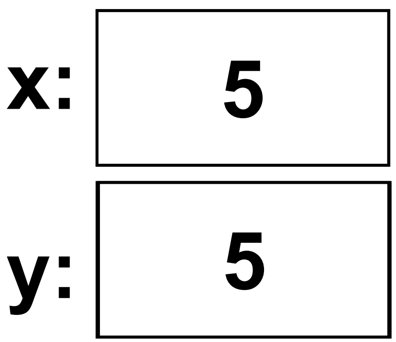
- The Java programming language does not specify how soon a finalizer will be invoked
- It is the Garbage collector that decides when the finalize method is invoked
- *finalize()* is never invoked more than once by a Java virtual machine for any given object
- For this reason, you should avoid finalizers
- Write instead a method on the class for this
  - and invoked it whenever an object is no longer needed

- Finalization has been **deprecated**

Primitive and Reference Data Type

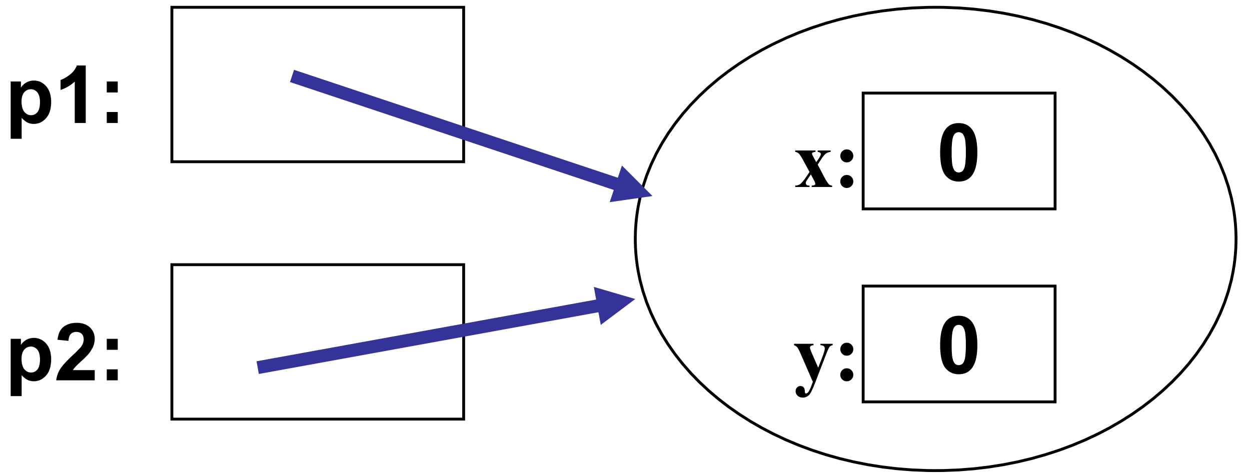
Primitive Data Type

```
int x, y;
x = 5;
y = x;
```



Reference Data Type

```
Point p1, p2;
p1 =  new Point();
p2 = p1;
```



Example

Point.java

```
public class Point {
   public int _x;
   public int _y;


    int getX() {
         return _x;
    }

   int getY() {
         return _y;
   }

   void move(int dx,
               int dy) {
         _x += dx;
         _y += dy;
   }
    // …
}
```

Main.java

```
public class Main {
   public static void main(String[]
                              args) {
         Point p1 = new Point();
         Point p2 = p1;

         int x = p1._x;
         // value of x?   0
         x = 4;
         // value of p1.x?  0
         p2._y = 5;
         // value of p2? (0, 5)    p1? (0, 5)
         p2.move(3, 3);
         // value of p2? (3, 8)    p1? (3, 8)
         p2 = new Point();
         // value of p2? (0, 0)    p1? (3, 8)
   }
}
```

Comparing Object References

int x = 5;

int y = 5;

- Result of (x == y)?
  - true

Point p1 = new Point(5, 6);

Point p2 = new Point(5, 6);

- Result of (p1 == p2)?
  - false
- Solution?
  - Add comparison method

```
public boolean equals(Point p) {
    return (p != null) &&
           (_x == p._x) &&
           (_y == p._y);
}
```

The *this* Keyword

- Sometimes, it is necessary to know the reference of the invoked object
  - Example: Comparison similar to the previous one, but returns the point with the greater x coordinate.
- How to do this?
  - How to know the invoked reference
  - **Must** use the this keyword
- Within an instance method (or a constructor), this is a reference to the invoked object

```
public Point greaterX(Point p) {
    if (p != null) && (_x < p._x))
        return p;
    return this;
 }
```

Method Signature

- Each method is identified by its *signature*
- Method signature: the method's name and its parameter types
  - The return type is not considered
- Each method in a class must have a distinct signature

Method Overloading

- A class can have more than one method with the same name as long as they have different parameter list
  - Meaning have a distinct signature

```
public class Pencil {
        . . .
        public void setPrice(float newPrice) {
                …;
        }

        public void setPrice(Pencil p) {
                 …;
        }
}
```

- How does the compiler know which method you're invoking?
  - Compares the number and type of the parameters and uses the matched one
  - If several candidates – use the most specific one
- Return type is not used to distinguish methods

Methods – Parameter Values

- Parameters are always passed by value

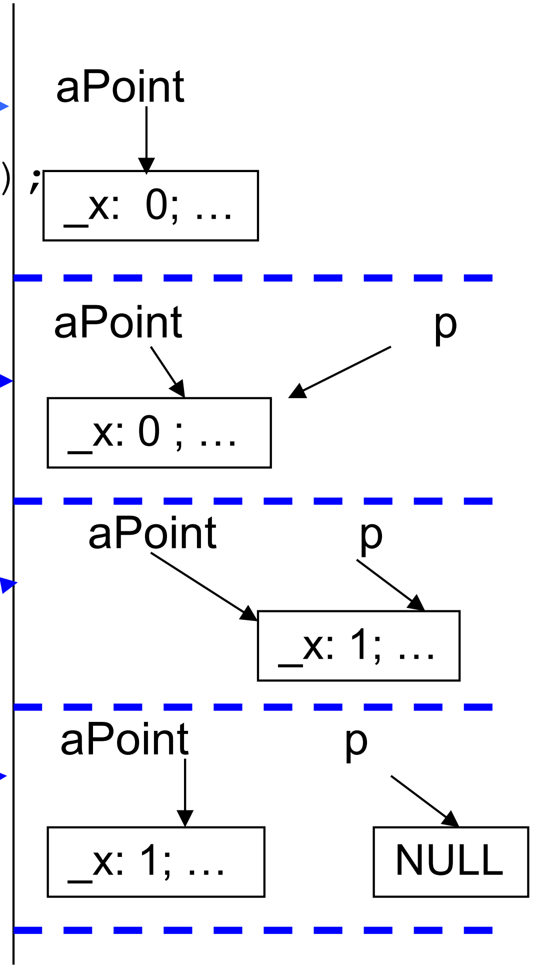
```
public void method1 (int a) {
    a = 6;
}

public void method2 ( ) {
    int b = 3;
    method1(b);    // now b = ?
                   // b = 3
}
```

- When the parameter is an object reference, it is the object reference, not the object itself, getting passed

☹ Haven't you said it's passed by value, not reference ?

<u>Another example</u>: (parameter is an object reference)

```
class PassRef{
  public static void main(String[] args) {
    Point aPoint = new Point();
    System.out.println("original x: " + aPoint.getX);


    changeX(aPoint);

    System.out.println("new x: " + aPoint.getX);
  }

  public static void changeX(Point p) {
    p.setX(1);
    p = null;
  }
}
```



- If you change any field of the object which the parameter refers to, the object is changed for every variable which holds a reference to this object

- You can change which object a parameter refers to inside a method without affecting the original reference which is passed

- What is passed is the object reference, and it is passed in the manner of “PASSING BY VALUE”!

Arrays in Java

- An *array* is a container object that holds a fixed number of values of a single type

- The length of an array is fixed
  - Defined when the array is created
- Declaration of an array:
  - Type[] nameOfArray;
  - Type nameOfArray[];
  - Note: declaration of an array just allocates memory for variable nameOfArray

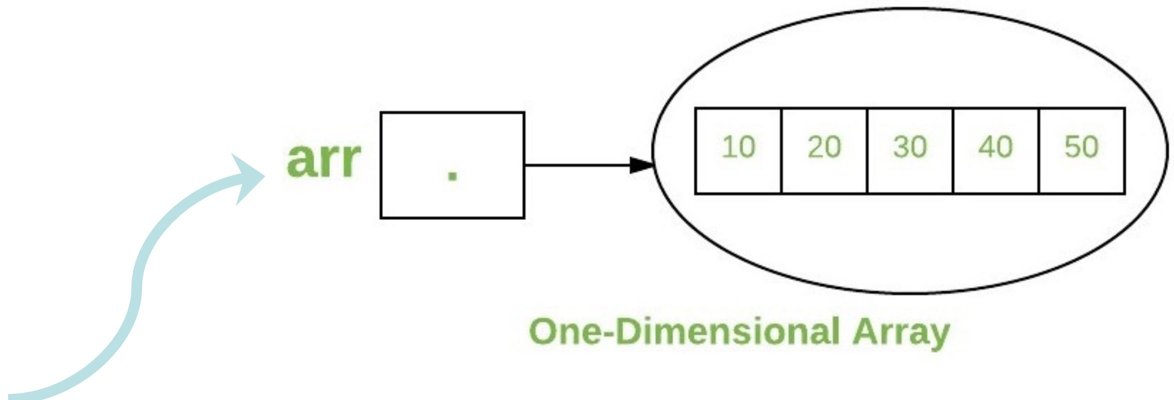
Creating an Array - 1

- There are three ways of creating an array

1. Instantiation of Array uses operator **new**
   - nameOfArray = new type[size];
     - Size must be of int type
   - Allocates memory for holding all elements of the array
   - The elements in the array allocated by *new* will automatically be initialized to
     - **zero** (for numeric types)
     - **false** (for boolean)
     - **null** (for reference types).

Creating an Array - 2

2. new ElementType[]{ value0, value1, … }



int[] arr = new int[]{ 10, 20, 30, 40, 50 };

– Anonymous array in Java

printArray(**new int**[]{10, 22, 44, 66}); //passing anonymous array to method

2. { elementValue0, elementValue1, … }

int[] arr = { 10, 20, 30, 40, 50 };

Arrays in Java

- An array is an object
- Size of an array is stored in *field* **length** of array
  - nameOfArray.length
- An array can have a size of 0
  - Similar to empty string

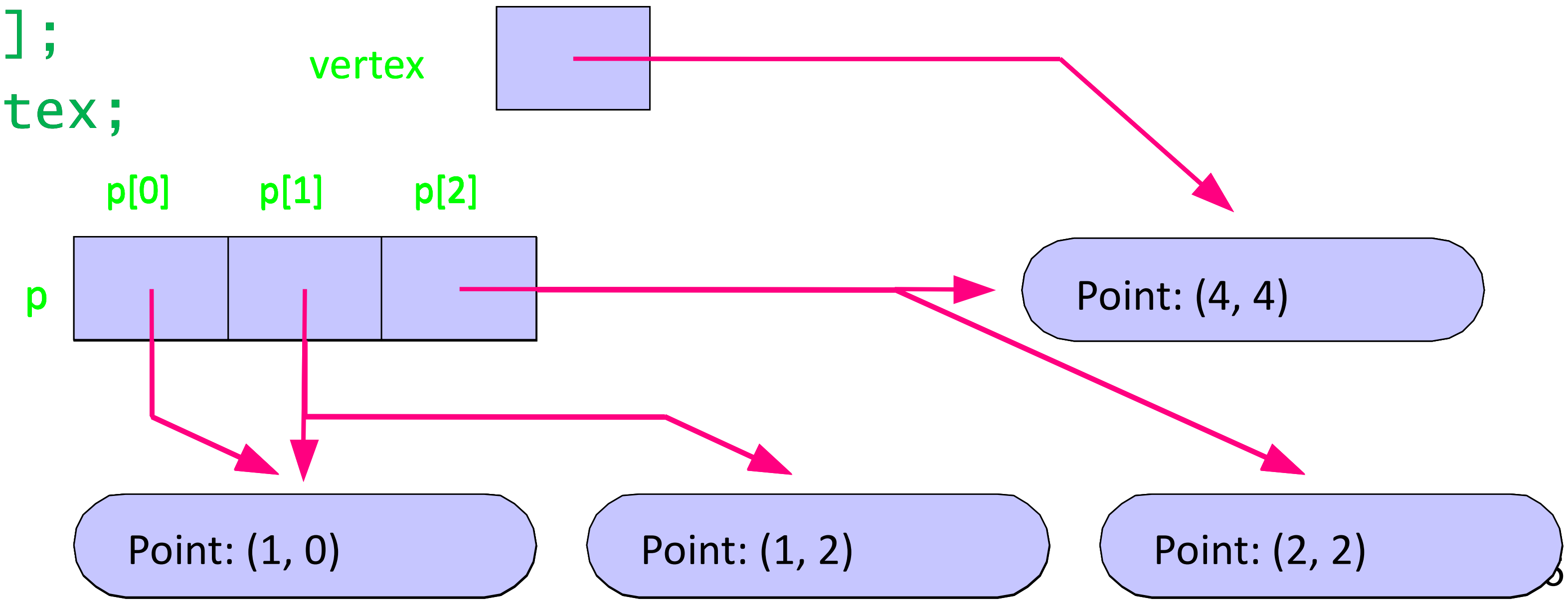
Accessing Java Array Elements

- Automatic bounds checking
  - Ensures any reference to an array element is valid
- Access to the elements of an array is checked by the JVM
  - Can only access [0, length -1]
  - Invalid accesses represented by ArrayIndexOutOfBoundsException

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 8
        at ClassName.main(ClassName.java:10)
```

Consider

```
Point[] p = new Point[3];
p[0] = new Point(0, 0);
p[1] = new Point(1, 1);
p[2] = new Point(2, 2);
p[0].setX(1);
p[1].setY(p[2].getY());
Point vertex = new Point(4,4);
p[1] = p[0];
p[2] = vertex;
```

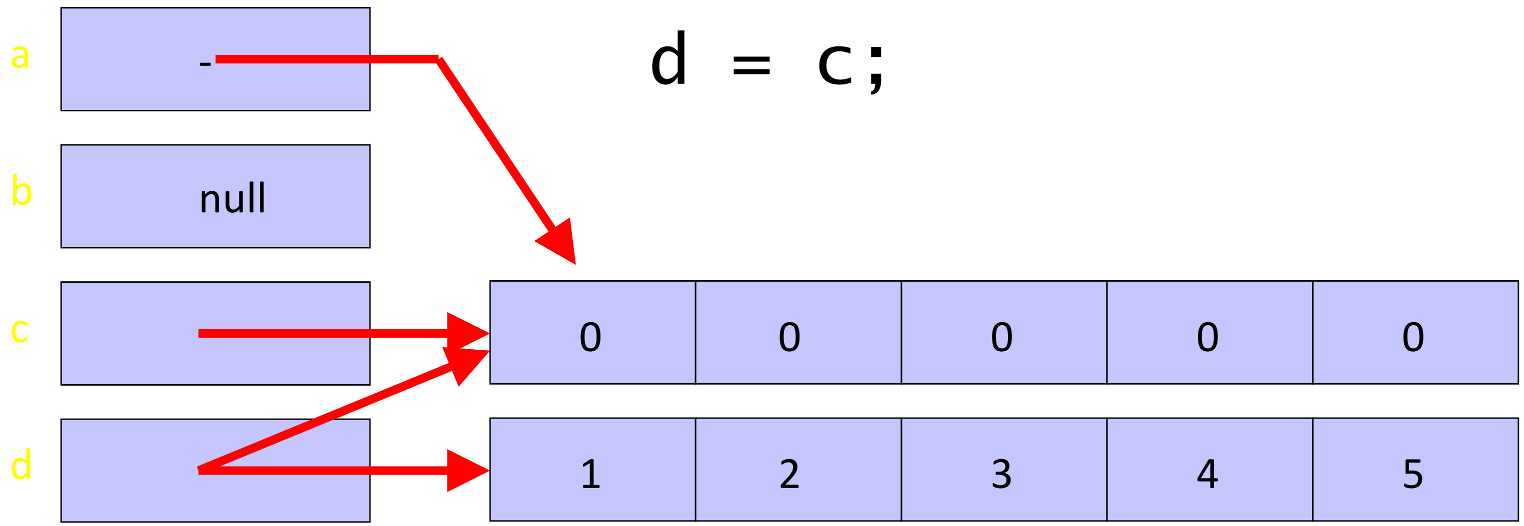


More about how Java represents Arrays

- Consider

```
int[] a;
int[] b = null;
int[] c = new int[5];
int[] d = { 1, 2, 3, 4, 5 };
a = c;
d = c;
```


int[] a;
int[] b = null;
int[] c = new int[5];
int[] d = {1,2,3,4,5};
a = c;
d = c;



Iterating over an Array

- Two ways
- Classical way

```
int[] arr = new int[20];
//…
for (int i = 0; i < arr.length; i++)
  System.out.println("Element at index " + i + " : "+ arr[i]);
```

- Loop for-each

```
for (type var : array) {
  statements using var;
}
```

```
int[] arr = new int[20];
//…
for (int v : arr)
  System.out.println("Element : ” + v);
```

Limitations of for-each Loop

- Cannot modify the array
- Do not keep track
- Only iterates forward over the array in single steps of index
- Cannot iterate over two arrays at the same time

Extra Information

Explicit initialization

- Syntax

id references an array of n elements. id[0] has value exp_0, id[1] has value exp_1, and so on.

$$\mathit{ElementType}[\,]\quad \mathit{id} = \{\quad exp_0\;,\; exp_1\;,\;\ldots\; exp_{n\text{-}1}\;\}\;;$$

Each exp_i is an expression that evaluates to type ElementType

Explicit initialization - Example

- Example
  - `String[] p = { “PO”, “is”, “Great”, “!” };`
  - `int[] unit = { 1 };`

- Equivalent to

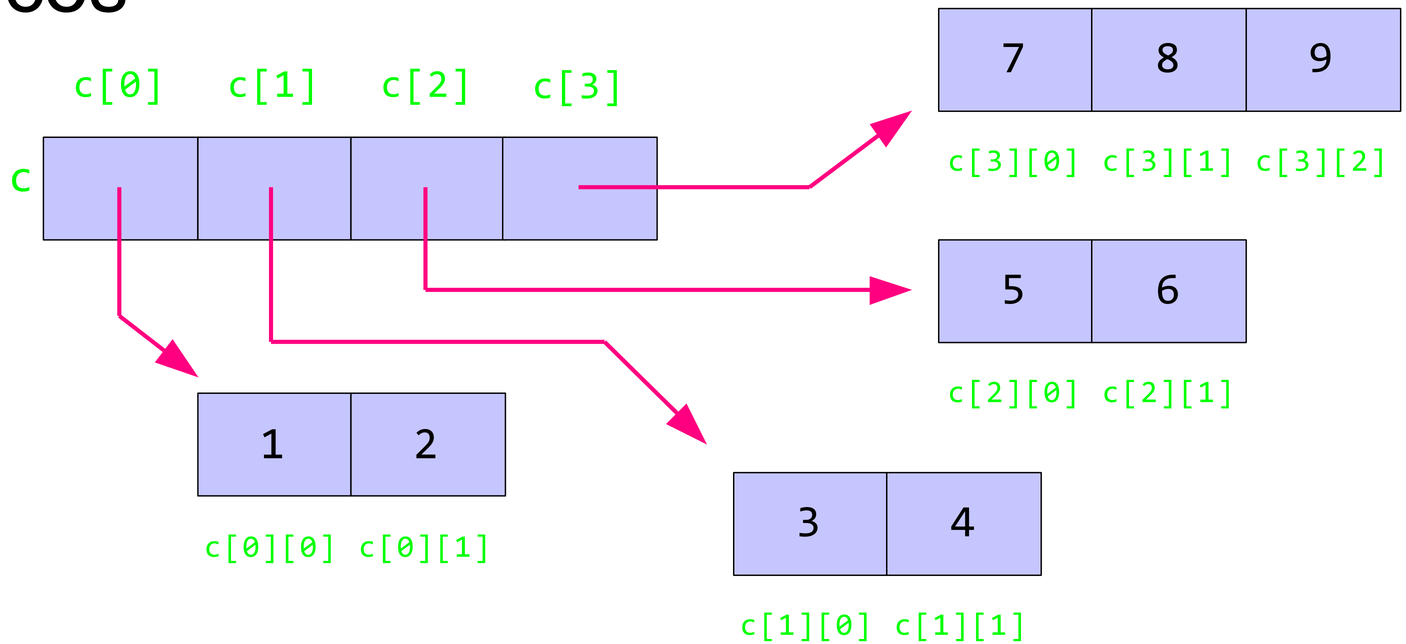
```
String[] p = new String[4];
p[0] = “PO”;    p[1] = “is";
p[2] = “Great”; p[3] = “!";

int[] unit = new int[1];
unit[0] = 1;
```

Explicit Initialization – Multidimensional arrays

- Segment

```
int c[][] = {{1, 2}, {3, 4}, {5, 6}, {7, 8, 9}};
```

- Produces

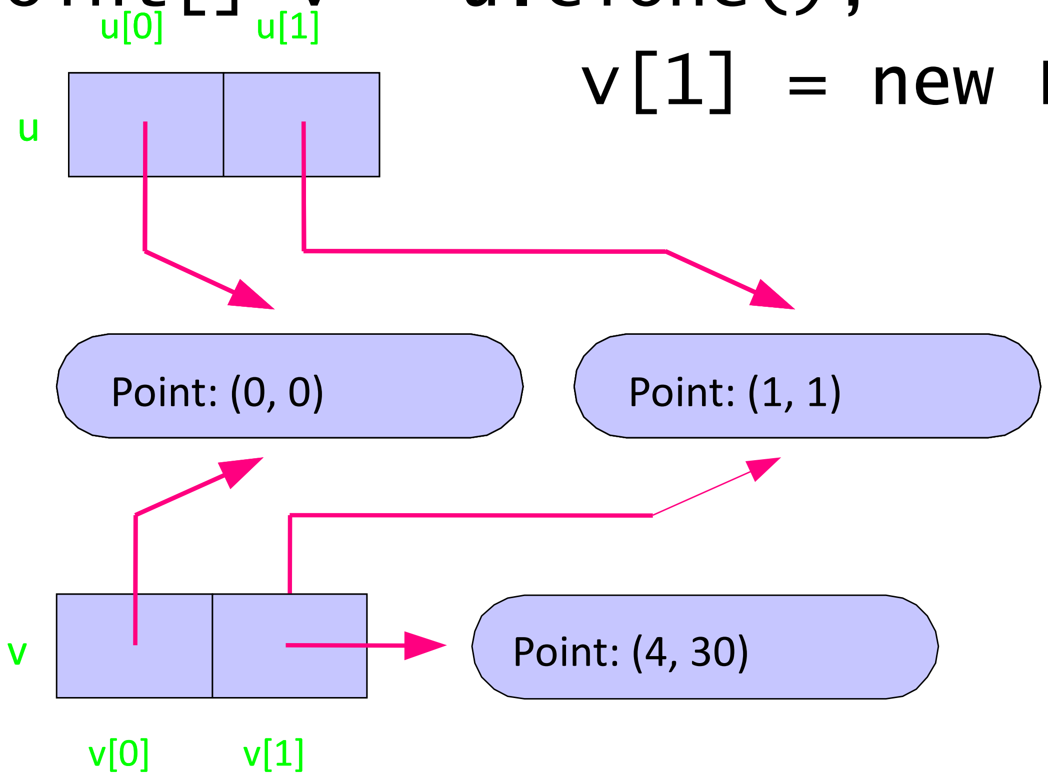


Array members

- Member clone()
  - Produces a shallow copy

```
Point[] u = { new Point(0, 0), new
  Point(1, 1)};
Point[] v = u.clone();
```

```
v[1] = new Point(4, 30);
```



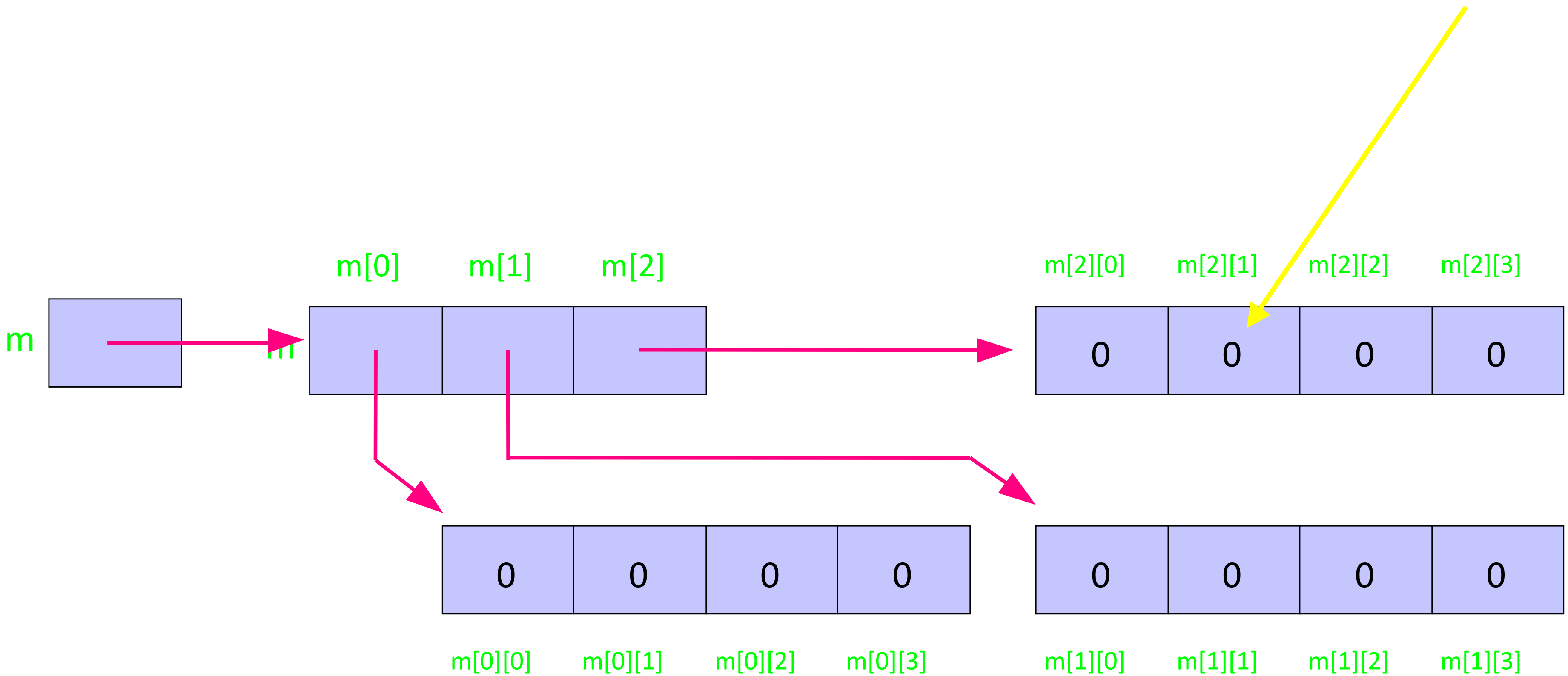
Example

- Segment

```
int[][] m = new int[3][4];
```

- Produces

When an array is created, each value is initialized!



Example – Sparse matrix

- Segment

```
String[][] s = new String[4][];
s[0] = new String[2];
s[1] = new String[2];
s[2] = new String[4];
s[3] = new String[3];
```

- Produces