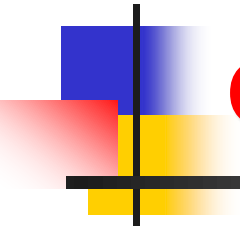
ADVANCED PROGRAMMING



OBJECT ORIENTED PROGRAMMING

CLASSES AND OBJECT

Outline

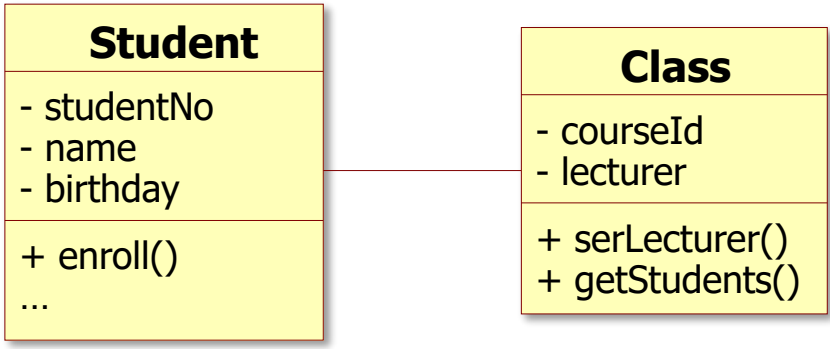
- Working with Classes and Objects
  - Defining Classes
  - Creating Objects
  - Writing and Invoking Constructors
- Using Methods
  - Defining a Method
  - The Static Methods and Variables
  - Methods with a Variable Number of Parameters
  - JavaBeans Naming Standard for Methods
- Method Overriding and Overloading

Outline

- Inheritance
- Abstract Classes
- Writing and Using Interfaces
- Object-Oriented Relationships
  - The is-a Relationship
  - The has-a Relationship
- Polymorphism
- Conversion of Data Types
- Understanding Garbage Collection

What is Object-Oriented Programming?

- OOP
  - Map your problem in the real world: Real world objects and actions match program objects and actions
  - Define "things" (objects) which can do something
  - Create a "type" (class) for these objects so that you don't have to redo all the work in defining an objects properties and behavior



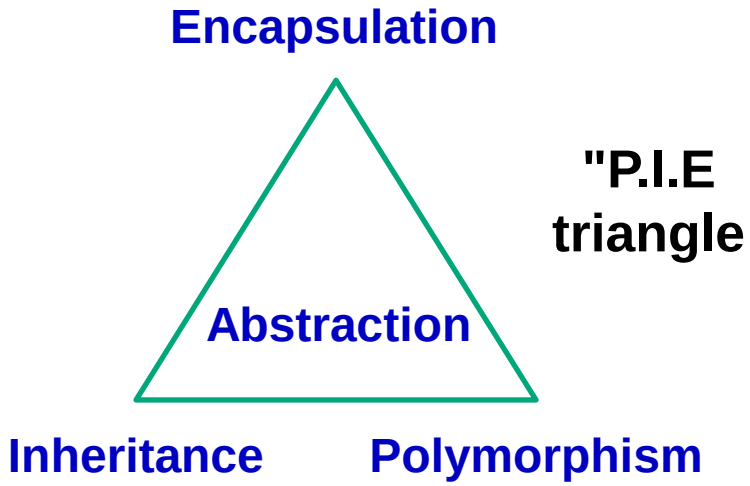
- An OO program: "*a bunch of objects telling each other what to do by sending messages*". (Smalltalk)
- A strong reflection of software engineering
  - Abstract data types
  - Information hiding (encapsulation)

Goals of Object Technology

- To create a software:
  - **Robustness (mạnh mẽ)**: capable of handling unexpected inputs that are not explicitly defined for its application.
  - **Adaptability (thích ứng)**: evolve over time in response to changing conditions in its environment.
  - **Reusability (tái sử dung)**: the same code should be usable as a component of different systems in various applications.

Important OO concepts

- Abstraction
- Objects & Class
  - Object state and behavior
  - Object identity
  - Messages
- Encapsulation
  - Information/implementation hiding
- Inheritance
- Polymorphism



"P.I.E triangle

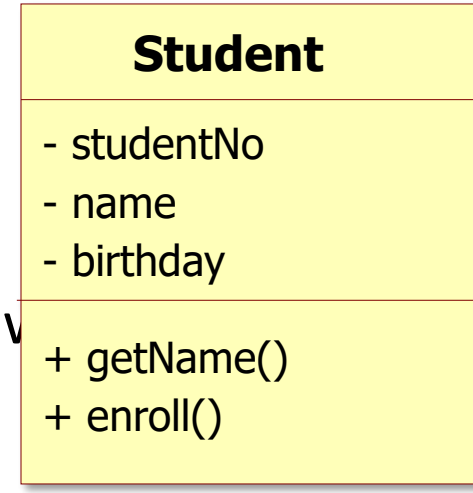
Benefits of Object Technology

1) **Faster** application **development** at a lower cost
2) **Decreased** maintenance **time**
3) **Less complicated** and faster customization
4) **Higher quality** code

Objects

- Objects are: Java is OOP. The core concept of the object-oriented approach is to break complex problems into smaller objects.
- An object is any entity that has attributes and methods
- An object possesses:
  - **Identity**: Định danh
    - A means of distinguishing it from other objects
  - **State**: Trạng thái: represents the data (value) of an object.
    - What the object remembers
  - **Interface**: Giao tiếp
    - Messages the object responds to
  - **Behavior**: Ứng xử represents the behavior

(functionality) of an object such as deposit, withdrav
    - What the object can do



Object Example

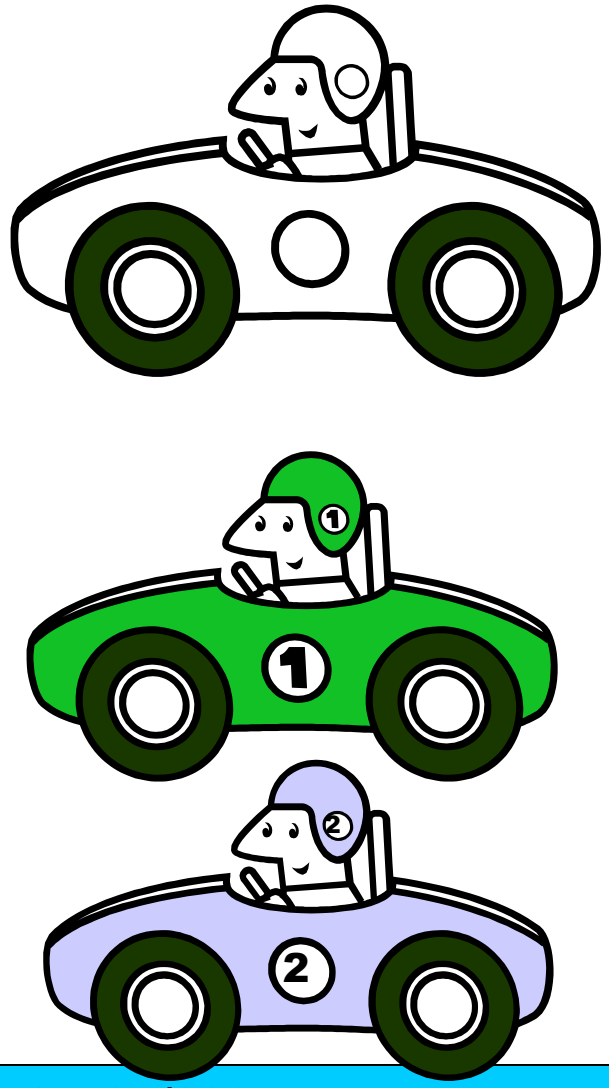
- The car shown in the figure can be considered an object.
  - It has an ID ("1"),
  - state (its color, for instance, and other characteristics),
  - an interface (a steering wheel and brakes, for example)
  - and behavior (the way it responds when the steering wheel is turned or the brakes are applied).

Classes

- A class
  - Defines the characteristics and variables common to all objects of that class
- Objects of the same class are similar with respect to:
  - Interface
  - Behavior (method)
  - State (variable)
- Used to instantiate (create an instance of) specific objects
- Provide the ability of reusability
  - Car manufacturers use the same *blueprint* to build many cars over and over

Class Example

- The car at the top of the figure represents a class
  - Notice that the ID and color (and presumably other state details) are not known, but the interface and behavior are.
- Below the "class" car are two objects which provide concrete installations of the class



Working with Classes

- The class is the basis for object-oriented programming
- The data and the operations on the data are encapsulated in a class
- A class is a **template** that contains the data variables and the methods that operate on those data variables following some logic
- All the programming activity happens inside classes

Working with Classes (cont.)

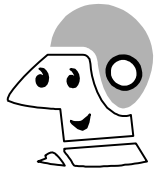
- The **data variables** and the methods in a class are called **class members**
- **Variables**, which hold the data (or point to it in case of reference variables), are said to represent the **state of an object**
- The **methods** constitute **class' behavior**

Message and Object Communication

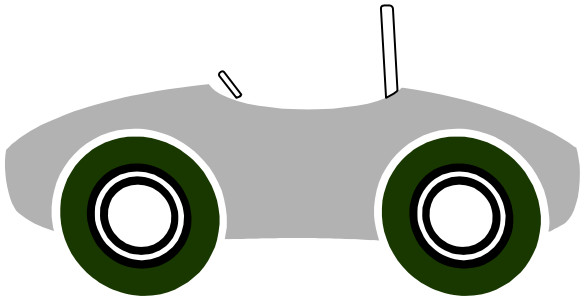
- Objects communicate via messages
- Messages in Java correspond to method calls (invocations)
- Three components comprise a message:
  1. The object to whom the message is addressed (**Your Car**)
  2. The name of the method to perform (**changeGears**)
  3. Any parameters needed by the method (**lower gear**)

Object Messaging Example

- By itself the car is incapable of activity. The car is only useful when it is interacted with by another object
- Object 1 sends a message to object 2 telling it to perform a certain action
- In other words, the driver presses the car's gas pedal to accelerate.



+



Accessing State

- State information can be accessed two ways:
- Using messages:
  - Eliminates the dependence on implementation
  - Allows the developer to hide the details of the underlying implementation

- "Accessor" messages are usually used instead of accessing state directly
  - Example: `getSpeed()` may simply access a state value called "speed" or it could hide a calculation to obtain the same value

Encapsulation

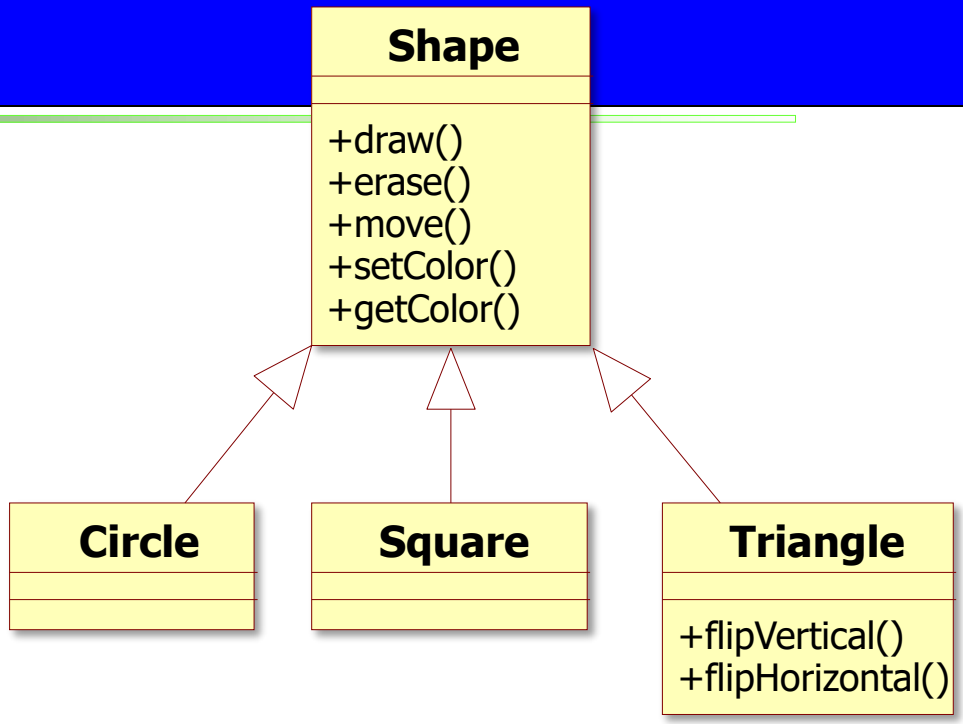
- Encapsulation: to group related things together, so as to use one name to refer to the whole group.
  - Functions/procedures encapsulate instructions
  - Objects encapsulate data and related procedures

Information hiding

- Information hiding: encapsulate to hide internal implementation details from outsiders
  - Outsiders see only interfaces
  - Programmers have the freedom in implementing the details of a system.
  - The only constraint on the programmer is to maintain the interface
  - **`public`**, **`private`**, and **`protected`**

Inheritance

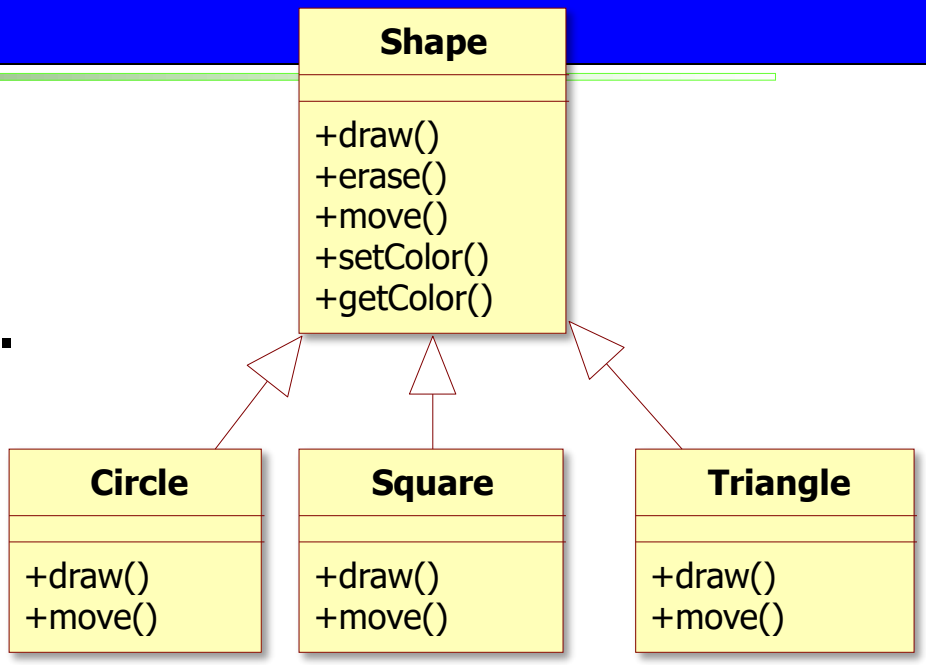
- "is-a" relations
- The general classes can be specialized to more specific classes

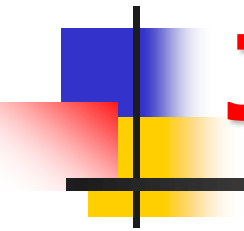


- Reuse of interfaces & implementation
- Mechanism to allow *derived* classes to possess attributes and operations of *base* class, as if they were defined at the *derived* class
- We can design generic services before specialising them

Polymophism

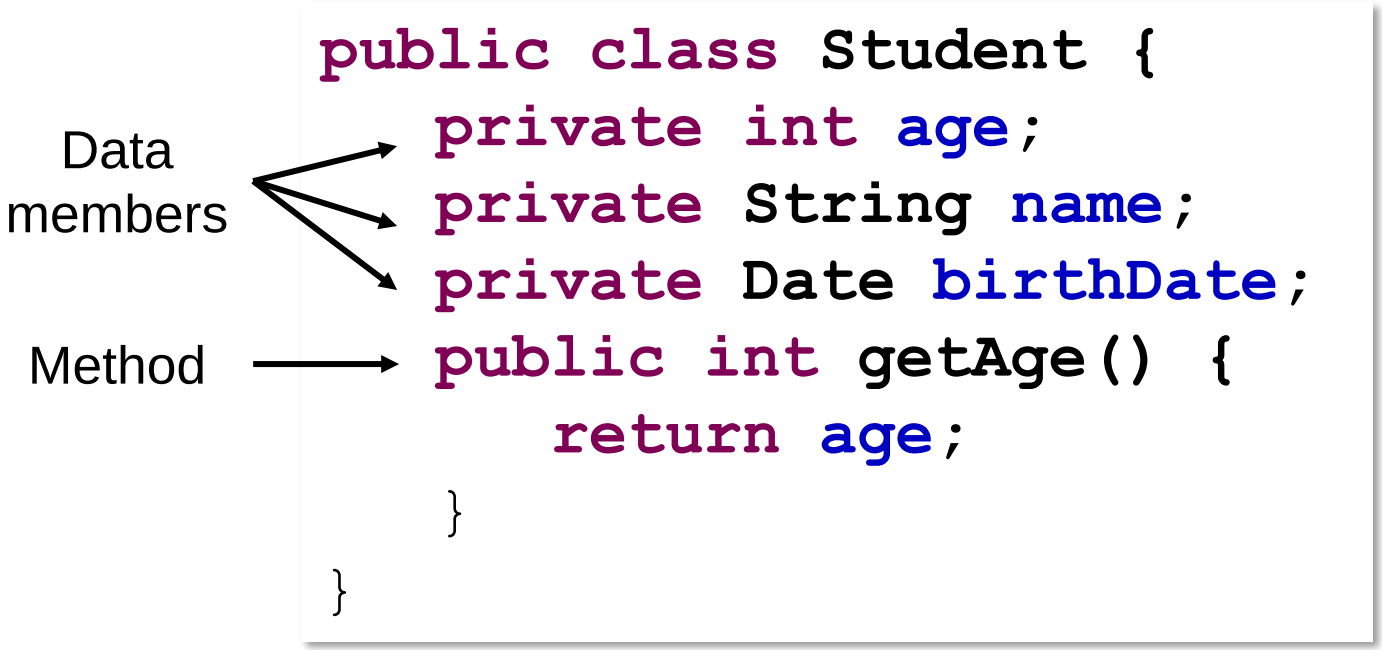
- Polymorphism:
  - Ability to assume different forms or shapes.
  - To exist in more than one form
- Object polymorphism:
  - Objects of different derived classes can be treated as if they are of the same class – their common base class
  - Objects of different classes understand the same message in different ways
    - example: on receiving message **`draw()`**, **`Rectangle`** and **`Triangle`** objects perform different **`draw()`** methods





Java Class

Java Classes



Class **Student** with data members and an instance method (accessor)

Classes

- Encapsulate **attributes** (fields) and **behavior** (methods)
  - Attributes and behavior are members of the class
- Members may belong to either of the following:
  - The whole class
    - **Class variables** and **methods**, indicated by the keyword `static`
  - Individual objects
    - **Instance variables** and **methods**
- Classes can be
  - Independent of each other (ko liên quan/độc lập)
  - Related by ***inheritance*** (superclass / subclass)
  - Related by ***type*** (interface)

Working with Objects

- Objects of pre-defined classes must be explicitly created by `new` operator
  - Allocate dynamic memory in heap memory
  - A **constructor** will be called to initialize the newly created object.
- Objects are manipulated via *references*
- Invoke object's methods:

```
<object reference>.<method_name>(<arguments>)
```

```
public class StringTest {
   public static void main(String args[]) {
      String s1 = new String("Hello, ");
      String s2 = s1.concat("world!");
      System.out.println("Here is the greeting" + s2);
   }
}
```

Defining a Class

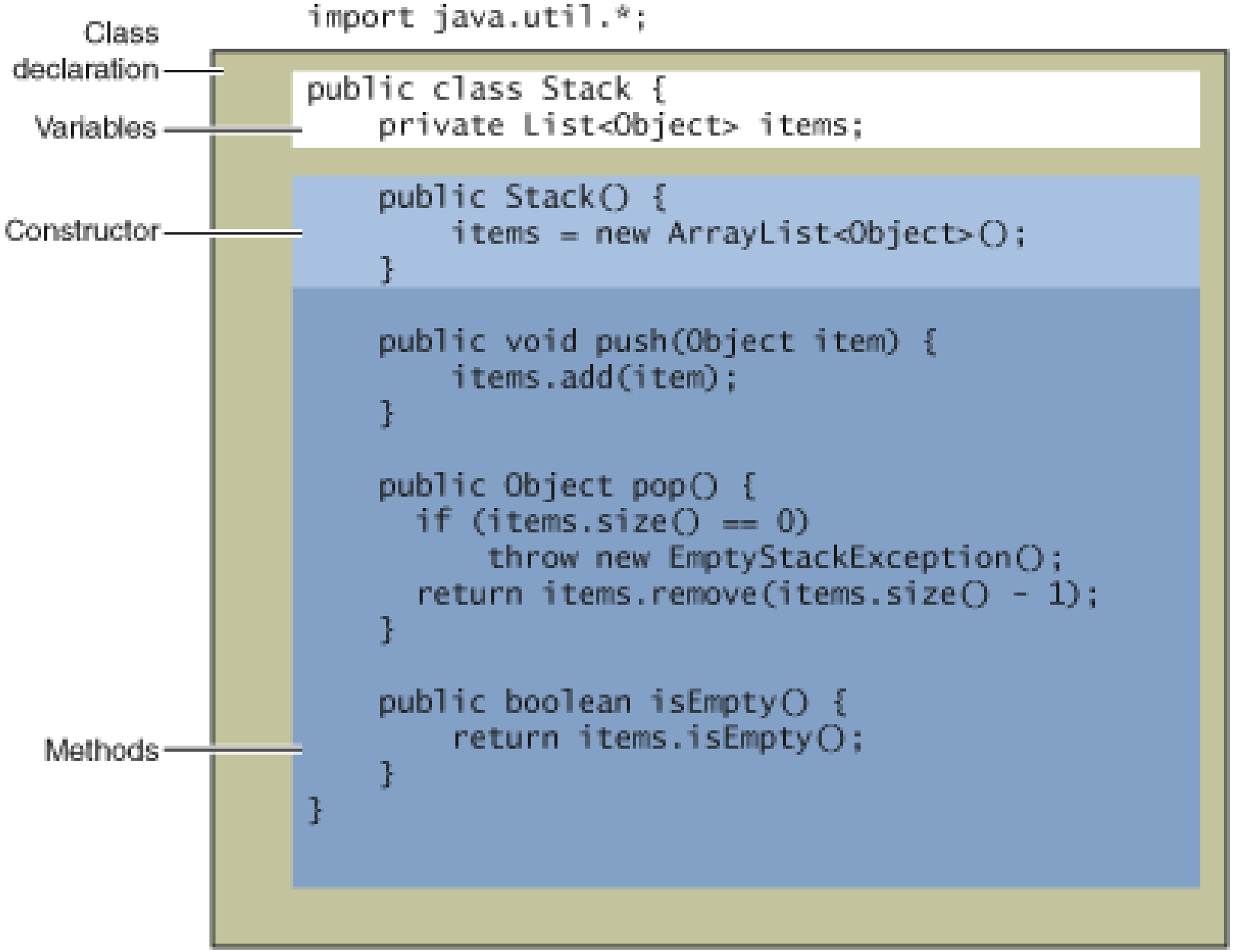
- A class declaration specifies a type
  - The identifier: specifies the name of the class
  - Attributes, methods, and access control
- Attributes:
  - object's instance variables
- Methods:
  - tasks that the objects can do
- Access modifiers:
  - **public** : Accessible anywhere by anyone
  - **protected** : Accessible only to the class itself and to its subclasses or other classes in the same "package"
  - **private** : Only accessible within the current class
  - default (no keyword): accessible within the current package

```
public class BankAccount {
   private String ownerName;
   private double balance;
   public void getOwnerName() {
      return ownerName;
   }
...
```

Implementing Classes

- Classes are grouped into packages
  - A package contains a collection of logically-related classes
- Source code files have the extension `.java`
  - There is one public class per **`.java`** file
- A class is like a blueprint; we usually need to create an object, or instance of the class

The Elements of a class



Class Declaration

- A class declaration specifies a type
  - The identifier
    - Specifies the name of the class
  - The optional **`extends`** clause
    - Indicates the superclass
  - The optional **`implements`** clause
    - Lists the names of all the interfaces that the class implements

```
public class BankAccount extends Account
    implements Serializable, BankStuff {

    // class body
}
```

Declaring Classes

Class Declaration Elements	
Element	Function
@annotation	**(Optional) An annotation (sometimes called meta-data)**
public	**(Optional) Class is publicly accessible**
abstract	**(Optional) Class cannot be instantiated**
final	**(Optional) Class cannot be subclassed**
class *NameOfClass*	**Name of the class**
<TypeVariables>	**(Optional) Comma-separated list of type variables**
extends *Super*	**(Optional) Superclass of the class**
implements *Interfaces*	**(Optional) Interfaces implemented by the class**

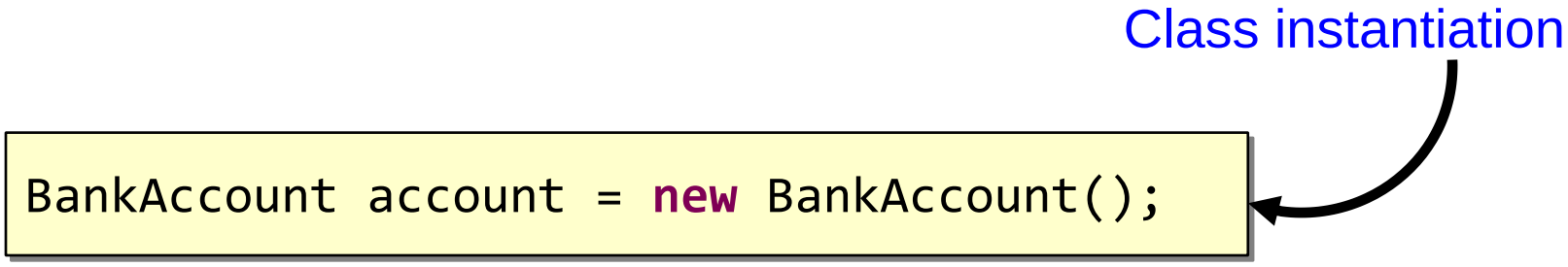
{
ClassBody
}
	Provides the class's functionality

Class Modifiers

- Class modifiers affect how the class can be used.
  - Examples: `public, abstract, final`
- `public` classes
  - May be accessed by any java code that can access its containing package
  - Otherwise it may be accessed only from within its containing package
- `abstract` classes
  - Can contain anything that a normal class can contain
    - Variables, methods, constructors
  - Provide common information for subclasses
  - Cannot be instantiated
- A class is declared `final` if it permits no subclasses.

Constructors

- A method that sets up a new instance of a class
  - The class body contains at least one constructor
- Use the **new** keyword with a constructor to create instances of a class



Writing and Invoking Constructors

- If you do not provide any constructor for a class you write, the compiler provides the default constructor for that class
- If you write at least one constructor for the class, the compiler does not provide a constructor
- In addition to the constructor (with no parameters), you can also define non-default constructors with parameters
  - From inside a constructor of a class, you can call another constructor of the same class
- You use the keyword **this** to call another constructor in the same class

Writing and Invoking Constructors

ComputerLab csLab = new ComputerLab();

- When the Java runtime system encounters this statement, it does the following, and in this order:
  1. Allocates memory for an instance of class ComputerLab
  2. Initializes the instance variables of class ComputerLab
  3. Executes the constructor ComputerLab()

More about Constructors

- Used to create and initialize objects
  - Always has the same name as the class it constructs (case-sensitive)
- No return type
  - Constructors return no value, but when used with new, return a reference to the new object

```
public class BankAccount {
   public BankAccount(String name)
      setOwner(name);
   }
}
```

Constructor definition

Constructor use

```
BankAccount account = new BankAccount("Joe Smith");
```

Default Constructors

- Default constructor
  - constructor with no arguments
- The Java platform provides a one only if you do not explicitly define any constructor
- When defining a constructor, you should also provide a default constructor

```
public class BankAccount {
    public BankAccount() {
    }

    public BankAccount(String name)
        setOwner(name);
    }
}
```

Overloading Constructors

- Overloading
  - When there are a number of constructors with different parameters
- Constructors are commonly overloaded to allow for different ways of initializing instances

```
BankAccount newAccount = new BankAccount();

BankAccount knownAccount =
          new BankAccount(accountNumber);
BankAccount namedAccount =
          new BankAccount("My Checking Account");
```

Constructor Example

- In a constructor, the keyword `this` is used to refer to other constructors in the same class

```
...
public BankAccount(String name) {
    super();
    owner = name;
}

public BankAccount() {
    this("TestName");
}

public BankAccount(String name, double initialBalance) {
    this(name);
    setBalance(initialBalance);
}
...
```

Constructor Chaining

- Constructor chaining
  - When one constructor invokes another within the class
- Chained constructor statements are in the form:
  `this(argument list);`
  - The call is only allowed once per constructor
  - It must be the first line of code
- Do this to share code among constructors

More on Constructor Chaining

- Superclass objects are built before the subclass

  `super (argument list)`

  - initializes superclass members
- The first line of your constructor can be:

  `super (argument list);`

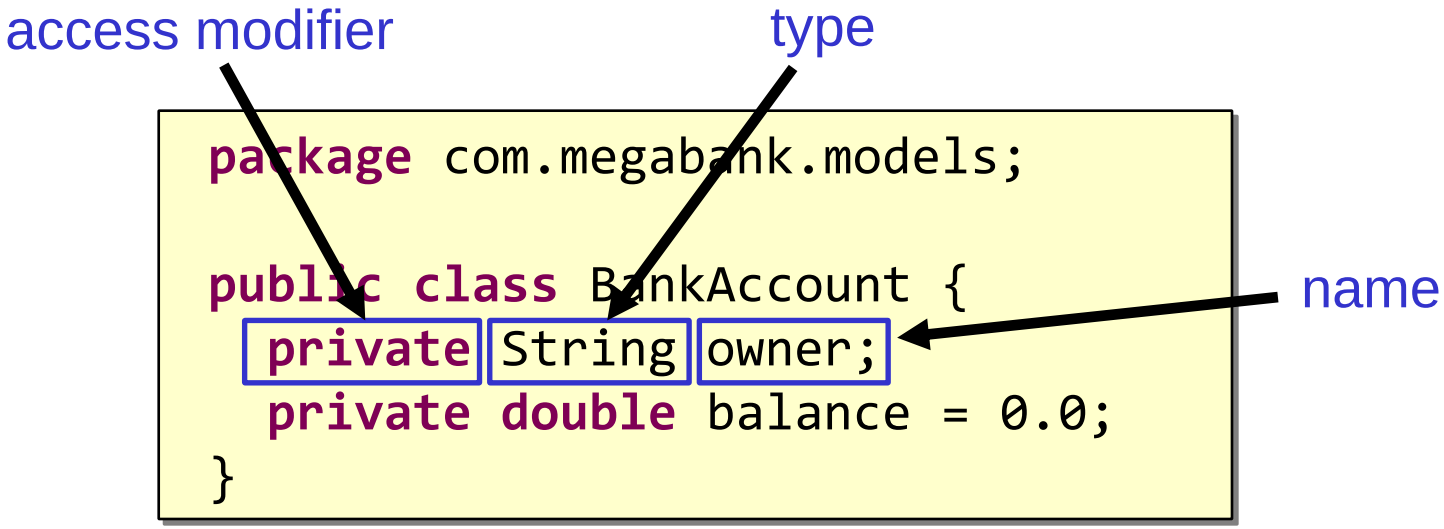
  `this (argument list);`
- You cannot use both `super()` and `this()` in the same constructor.
- The compiler supplies an implicit `super()` constructor for all constructors.

Java Destructors?

- Java does not have the concept of a destructor for objects that are no longer in use
- Deallocation is done automatically by the JVM
  - The garbage collector reclaims memory of unreferenced objects
  - The association between an object and an object reference is severed by assigning another value to the object reference, for example:
    - `objectReference = null;`
  - An object with no references is a candidate for deallocation during garbage collection

Declaring Member Variables - Fields

- Fields
  - Defined as part of the class definition
  - Objects retain state in fields
  - Each instance gets its own copy of the instance variables
- Fields can be initialized when declared
  - Default values will be used if fields are not initialized



Declaring Member Variables

Variable Declaration Elements	
Element	Function
accessLevel `public`, `protected`, `private`	(Optional) Access level for the variable
static	(Optional) Declares a class variable
final	(Optional) Indicates that the variable's value cannot change
transient	(Optional) Indicates that the variable is transient (should not be serialized)
volatile	(Optional) Indicates that the variable is volatile
type name	The type and name of the variable

Controlling Access to Members of a Class

Access Levels				
Specifier	Class	Package	Subclass	World
private	Y	N	N	N
no specifier	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

Encapsulation

- Private state can only be accessed from methods in the class
- Mark fields as private to protect the state
  - Other objects must access private state through public methods

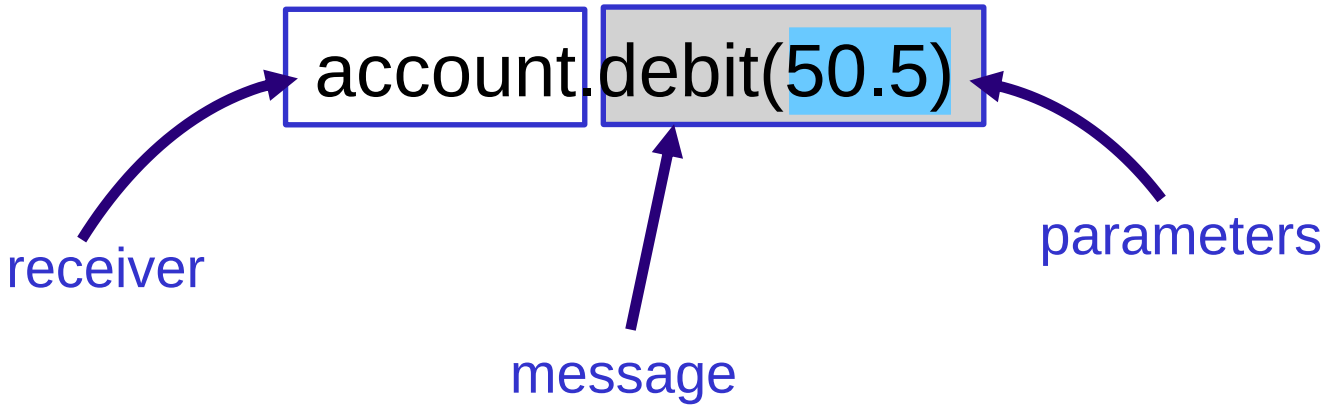
```
package com.megabank.models;

public class BankAccount {
  private String owner;
  private double balance = 0.0;
}
```

```
public String getOwner() {
  return owner;
}
```

Messages

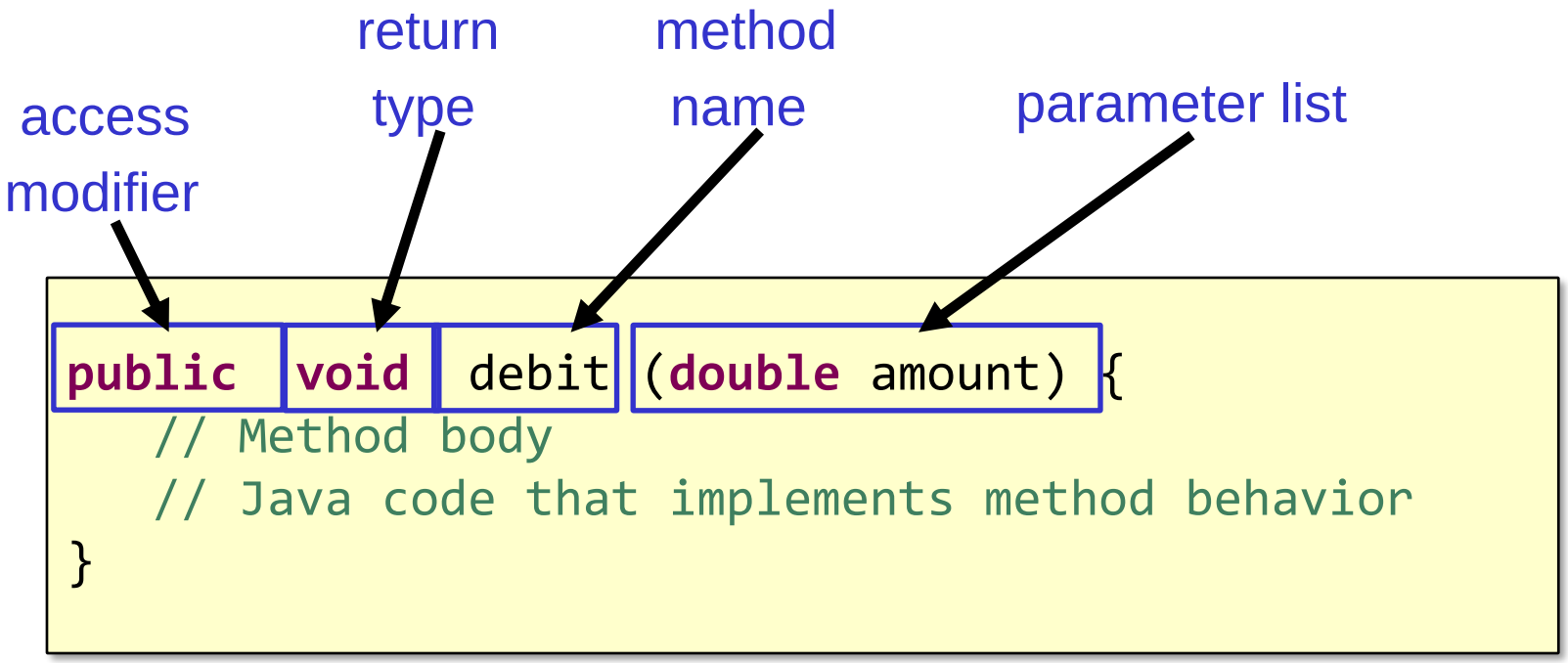
- Use messages to invoke behavior in an object

```
BankAccount account = new BankAccount();
account.setOwner("Smith");
account.credit(1000.0);
account.debit(50.5);
```



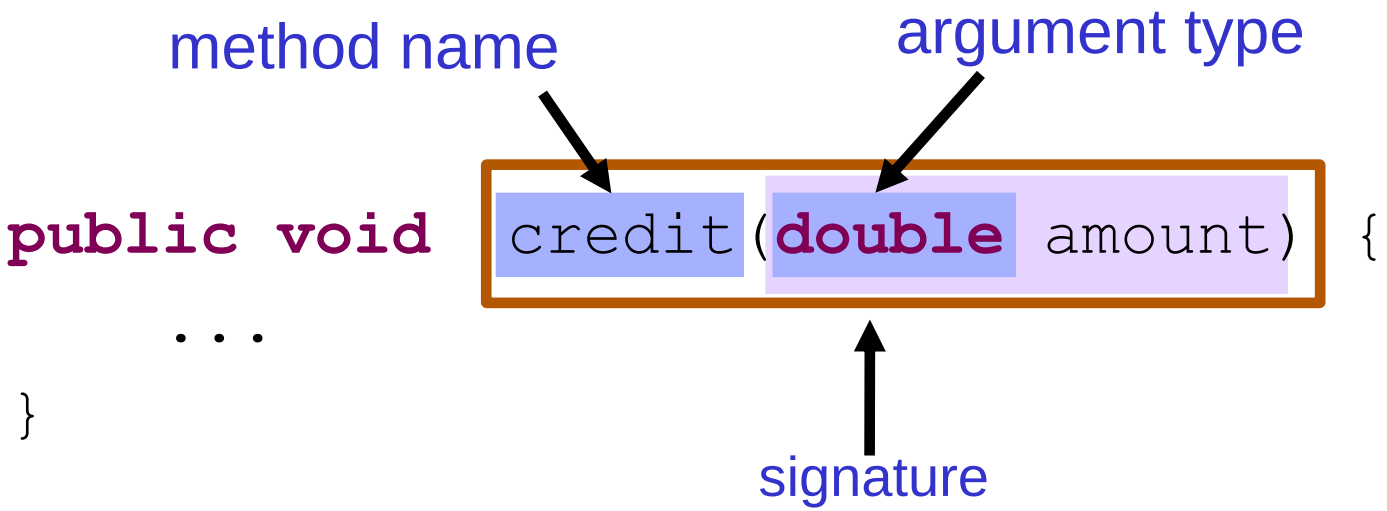
Methods

- Methods define
  - How an object responds to messages
  - The behavior of the class
    - All methods belong to a class



Method Signatures

- A class can have many methods with the same name
  - Each method must have a different signature
- The method signature consists of
  - The method name
  - Argument number and types



Method Parameters

- Arguments (parameters) are passed by
  - Value for primitive types
  - Object reference for reference types
- Primitive values cannot be modified when passed as an argument

```
public void method1() {
    int a = 0;
    System.out.println(a); // outputs 0
    method2(a);
    System.out.println(a); // outputs 0
}
```

```
void method2(int a) {
    a = a + 1;
}
```

Returning from Methods

- Methods return, at most, one value or one object
  - If the return type is void, the return statement is optional
- There may be several return statements
  - Control goes back to the calling method upon executing a return

```
public void debit(double amount) {
  if (amount > getBalance()) return;
  setBalance(getBalance() - amount);
}
```

```
public String getFullName() {
  return getFirstName() + " " + getLastName();
}
```

Invoking Methods

- To call a method, use the dot operator
  - The same operator is used for both class and instance methods
  - If the call is to a method of the same class, the dot operator is not necessary

```
BankAccount account = new BankAccount();
account.setOwner("Smith");
account.credit(1000.0);
System.out.println(account.getBalance());
...
```

BankAccount method

```
public void credit(double amount) {
   setBalance(getBalance() + amount);
}
```

Method Overriding

- **Method Overriding**
  - method overriding is a feature of Java that lets the programmer declare and implement a method in a subclass that has the same signature as a method in the superclass
- **Rules for method overriding**
  - You cannot override a method that has the final modifier.
  - You cannot override a static method to make it non-static.
  - The overriding method and the overridden method must have the same return type.
  - The number of parameters and their types in the overriding method must be same as in the overridden method and the types must appear in the same order. However, the names of the parameters may be different.

Method Overriding

- **Rules for method overriding (cont.)**
  - You cannot override a method to make it less accessible.
  - If the overriding method has a throws clause in its declaration, then the following two conditions must be true:
    - The overridden method must have a throws clause, as well.
    - Each exception included in the throws clause of the overriding method must be either one of the exceptions in the throws clause of the overridden method or a subclass of it.
  - If the overridden method has a throws clause, the overriding method does not have to.

Overloading Methods

- Signatures permit the same name to be used for many different methods
  - Known as overloading
- Two or more methods in the same class may have the same name but different parameters
- The `println()` method of `System.out.println()` has 10 different parameter declarations:
  - `boolean`, `char[]`, `char`, `double`, `float`, `int`, `long`, `Object`, `String` and one with no parameters
  - You don't need to use different method names, for example "`printString`", "`printDouble`", ...

Overriding

- Override a method when a new implementation in a subclass is provided, instead of inheriting the method with the same signature from the superclass

```
public class BankAccount {
    private float balance;
    public int getBalance() {
        return balance;
    }
}
public class InvestmentAccount extends BankAccount{
    private float cashAmount
    private float investmentAmount;
    public int getBalance() {
        return cashAmount + investmentAmount;
    }
}
```

Variable – Length Arguments

- The printf method takes any number of arguments
  - You could use overloading to define a few versions of printf with different argument lengths, but it takes any number of arguments
- To do this yourself, use `"type ... variable"`
  - variable becomes an array of given type
  - Only legal for final argument of method
  - Examples
    - `public void printf(String format, Object ... args)`
    - `public int max(int ... numbers)`
      - Can call max(1, 2, 3, 4, 5, 6) or max(someArrayOfInts)
- Use sparingly
  - You usually know how many arguments are possible

Methods with Variable Argument Lists (var-args)

- **Var-arg type** When you declare a var-arg parameter, you must specify the type of the argument(s) this parameter of your method can receive. (This can be a primitive type or an object type.)
- **Basic syntax** To declare a method using a var-arg parameter, you follow the type with an ellipsis (...), a space, and then the name of the array that willhold the parameters received.
- **Other parameters** It's legal to have other parameters in a method that uses a var-arg.
- **Var-args limits** The var-arg must be the last parameter in the method's signature, and you can have only one var-arg in a method.

Varargs: Example

```
public class MathUtils {
   public static int min(int ... numbers) {
      int minimum = Integer.MAX_VALUE;
      for (int number: numbers) {
         if (number < minimum) {
            minimum = number;
         }
      }
      return minimum;
   }

   public static void main(String[] args) {
      System.out.printf("Min of 2 nums: %d.%n", min(2,1));
      System.out.printf("Min of 7 nums: %d.%n",
                                 min(2,4,6,8,1,2,3));
   }
}
```

main Method

- An application cannot run unless at least one class has a main method
- The JVM loads a class and starts execution by calling the main(String[] args) method
  - **`public`**: the method can be called by any object
  - **`static`**: no object need be created first
  - **`void`**: nothing will be returned from this method

```
public static void main(String[] args) {
   BankAccount account = new BankAccount();
   account.setOwner(args[0]);
   account.credit(Integer.parseInt(args[1]));
   System.out.println(account.getBalance());
   System.out.println(account.getOwner());
}
```

More about method - Using Methods

- Methods represent operations on data and also hold the logic to determine those operations
- Using methods offer two main advantages:
  - A method may be executed (called) repeatedly from different points in the program
  - Methods help make the program logically segmented, or modularized. A modular program is less error prone, and easier to maintain

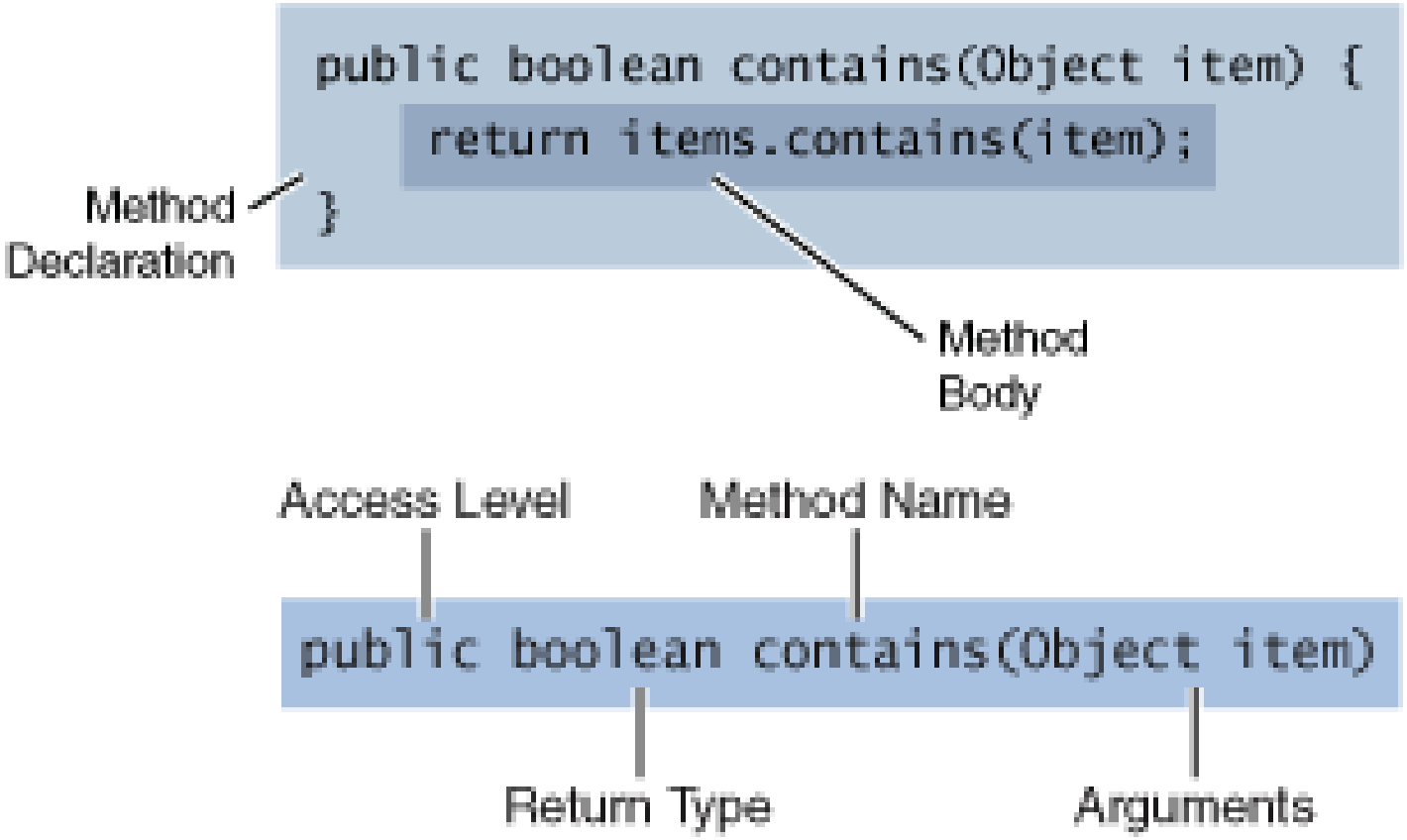
More about method - Defining a Method

- Defining a method in a program is called **method declaration**
- A method consists of the following elements:
  - **Name**: The name identifies the method and is also used to call (execute) the method. Naming a method is governed by the same rules as those for naming a variable
  - **Parameter(s)**: A method may have zero or more parameters defined in the parentheses
  - **Argument(s)**: The parameter values passed in during the method call
  - **Return type**: A method may optionally return a value as a result of a method call. Special **void** return type
  - **Access modifier**: Each method has a default or specified access modifier

Defining a Method (cont.)

- The method name and return type are mandatory in a method declaration
- Methods and variables visibility:
  - In a normal case, methods and variables visible only within an instance of the class, and hence each instance has its own copy of those methods and variables
  - Those that are visible from all the instances of the class. Those are called **static**

Defining Methods



- Access level: *public, protected, private*
  Default is *package private*

Defining Methods

Element	Function
@annotation	(Optional) An annotation
accessLevel	(Optional) Access level for the method
static	(Optional) Declares a class method
<TypeVariables>	(Optional) Comma-separated list of type variables.
abstract	(Optional) Indicates that the method must be implemented in concrete subclasses.
final	(Optional) Indicates that the method cannot be overridden
synchronized	(Optional) Guarantees exclusive access to this method
returnType methodName	The method's return type and name
(***paramList*** **)**	The list of arguments to the method
throws exceptions	(Optional) The exceptions thrown by the method

Example

- Tạo lớp phân số bao gồm:
  - Thành phần dữ liệu:
    - Tử số
    - Mẫu số
  - Phương thức
    - Constructor chuẩn, có tham số
    - In phân số
    - Rút gọn phân số
    - Cộng , trừ, nhân, chia 2 phân số
  - Biểu diễn dữ liệu cho 2 phân số bất kỳ để kiểm tra các chức năng trên.

Creating Objects

- Point originOne = new Point(23, 94);
- Rectangle rectOne = new Rectangle(originOne,100,200);
- Rectangle rectTwo = new Rectangle(50, 100);

Each statement has the following three parts:

1. Declaration: The code set in bold are all variable declarations that associate a variable name with an object type.
2. Instantiation: The new keyword is a Java operator that creates the object. As discussed below, this is also known as *instantiating a class*.
3. Initialization: The new operator is followed by a call to a constructor. For example, Point(23, 94) is a call to Point's only constructor. The constructor initializes the new object.

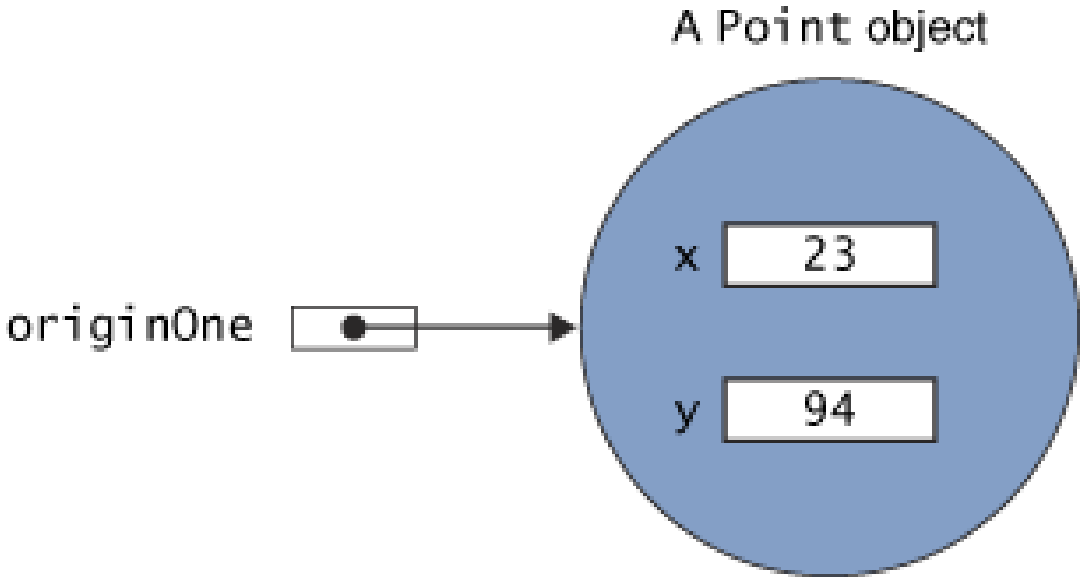
Declaring a Variable to Refer to an Object

type name

- This notifies the compiler that you will use *name* to refer to data whose type is *type*. The Java programming language divides variable types into two main categories: *primitive types*, and *reference types*.
- The declared type matches the class of the object:
  MyClass myObject = new MyClass(); or
  MyClass myObject;
- The declared type is a parent class of the object's class:
  MyParent myObject = new MyClass(); or
  MyParent myObject
- The declared type is an interface which the object's class implements:
  MyInterface myObject = new MyClass(); or
  MyInterface myObject
- A variable in this state, which currently references no object, is said to hold a *null reference*.

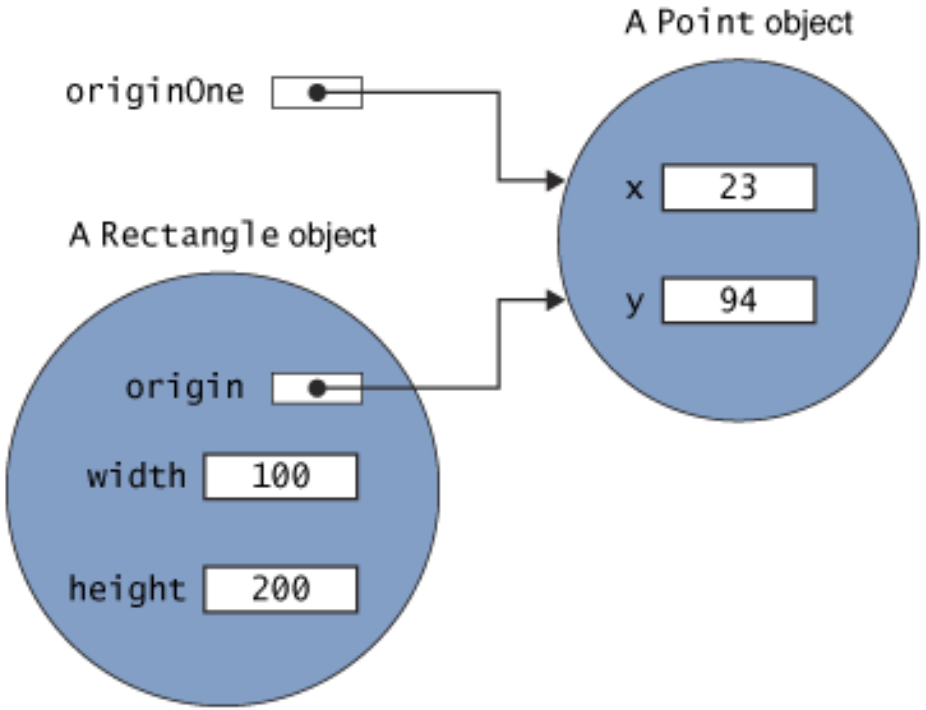
Initializing an Object

```
public class Point {
    public int x = 0;
    public int y = 0;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
Point originOne = new Point(23, 94);
```



Initializing an Object

```
public class Rectangle {
    public int width;
    public int height;
    public Point origin;
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
}
```




```
Rectangle rectOne = new
   Rectangle(originOne, 100, 200);
```

Using Objects

- You can use an object in one of two ways:
  - Directly manipulate or inspect its variables
  - Call its methods
- Referencing an Object's Variables
  The following is known as a *qualified name*:

```
objectReference.variableName
System.out.println("Width of rectOne: " +
                                        rectOne.width);
System.out.println("Height of rectOne: " +
                                        rectOne.height);
int height = new Rectangle().height;
```

- Calling an Object's Methods

```
objectReference.methodName(); or
objectReference.methodName(argumentList);
System.out.println("Area of rectOne: " +
                                        rectOne.area());
rectTwo.move(40, 72);
```

The Static Methods and Variables

- The **static** modifier may be applied to a variable, a method, and a block of code
- A static element of a class is visible to all the instances of the class
- If one instance makes a change to it, all the instances see that change
- A static variable is initialized when a class is loaded,
  - An instance variable is initialized when an instance of the class is created
- A static method also belongs to the class, and not to a specific instance of the class
  - Therefore, a static method can only access the static members of the class.

Static Members

- `Static` fields and methods belong to the class
  - Changing a value in one object of that class changes the value for all the objects
- `Static` methods and fields can be accessed without instantiating the class
- `Static` methods and fields are declared using the static keyword

```
public class MyDate {
   public static long getMillisSinceEpoch(){
       …
   }
}
...
long millis = MyDate.getMillisSinceEpoch();
```

The Static Methods

- A method declared static in a class cannot access the non static variables and methods of the class
- A static method can be called even before a single instance of the class exists
  - Static method main(), which is the entry point for the application execution
  - It is executed without instantiating the class in which it exists

Bài Tập

- Hãy định nghĩa lớp "kiến" (Ant):
  - Mỗi một đối tượng tạo ra từ lớp Ant này là 1 con kiến
  - Tất cả các con kiến tạo ra từ cùng lớp Ant này tạo ra một đàn kiến
  - giả sử kiến không chết
- Hãy thiết kế lớp Ant sao cho khi hỏi 1 con kiến bất kỳ trong đàn thì nó cũng cho ta biết tổng số con kiến có trong đàn

Example

```
class MyClass {
   String salute = "Hello";
    public static void main(String[] args){
       System.out.println("Salute: " + salute);
    }
}
```

Error: Cannot access a non static variable from inside a static method

Using Modifiers

- You cannot specify any modifier for the variables inside a method
- You cannot specify the protected modifier for a top-level class
- A method cannot be overridden to be less public.

Access Modifier	Class	Subclass	Package	World
private	Yes	No	No	No
protected	Yes	Yes	Yes	No
public	Yes	Yes	Yes	Yes
Default	Yes	No	Yes	No

Final Modifiers

- The **final** Modifier
  - The final modifier may be applied to a class, a method, or a variable. It means, in general, that the element is final
  - If the element declared final is a variable, that means the value of the variable is constant, and cannot be changed
  - If a class is declared final, it means the class cannot be extended
  - If a final method cannot be overridden

Final Modifiers

```
class Calculator {
  final int dime = 10;
  int count = 0;
  Calculator (int i) {
     count = i;
  }
}
class RunCalculator {
    public static void main(String[] args) {
        final Calculator calc = new Calculator(1);
        calc = new Calculator(2);
        calc.count = 2;
        calc.dime  = 11;
        System.out.println("dime: " + calc.dime);
   }
}
```

Error: calc is final

OK: default access

Error: dime is final

Final Members

- A final field is a field which cannot be modified
  - This is the java version of a constant
- Constants associated with a class are typically declared as **`static final`** fields for easy access
  - A common convention is to use only uppercase letters in their names

```
public class MyDate {
   public static final long
       SECONDS_PER_YEAR = 31536000;
   ...
}
...
long years = MyDate.getMillisSinceEpoch()/
   (1000 * MyDate.SECONDS_PER_YEAR);
```

Passing Arguments into Methods

- Assume you declare a variable in your method, and then you pass that variable as an argument in a method call
- The question is: What kind of effect can the called method have on the variable in your method?
- There are two kind:
  - pass-by-value
  - pass-by-reference

Passing a Primitive Variable

- When a primitive variable is passed as an argument in a method call, only the copy of the original variable is passed
- Any change to the passed variable in the called method will not affect the variable in the calling method
- It is called **pass-by-value**

Passing Primitive Variables

```
public class SwapNumber extends TestCase{
    public void swap(int a, int b){
      int c = a;
      a = b;
      b = c;
    }
  public void test(){
      int a = 1, b = 2;
      System.out.println("Before swap a: "+
                                a +  " , b: "+b);
      swap(a,b);
      System.out.println("After swap a: "+
                                a + " , b: "+b);
  }
}
```

Passing Primitive Variables

```
public class SwapNumber extends TestCase{
    public void swap(Integer a1, Integer b1){
     Integer c = a1;
     a1 = b1;
     b1 = c;
    }
  public void test(){
     Integer a = new Integer (1),
     b = new Integer (2);
     System.out.println("Before swap a: "+
                              a +  " , b: "+b);
     swap(a,b);
     System.out.println("After swap a: "+
                              a + " , b: "+b);
  }
}
```

Passing Object Reference Variables

- When you pass an object variable into a method, you must keep in mind that you're passing the object *reference,* and not the actual object itself.

```
import java.awt.Dimension;
class ReferenceTest {
  public static void main (String [] args) {
    Dimension d = new Dimension(5,10);
    ReferenceTest rt = new ReferenceTest();
    System.out.println("Before modify() d.height = "+
                                                  d.height);
    rt.modify(d);
    System.out.println("After modify() d.height = "+ d.height);
  }
  void modify(Dimension dim) {
    dim.height = dim.height + 1;
    System.out.println("dim = " + dim.height);
  }
}
```

Passing Object Reference Variables

```
Before modify()   d.height = 10
dim = 11
After modify()    d.height = 11
```

Passing Object Reference Variables

```
public class PassingVar extends TestCase{
    public void modify(Student st){
        st.setName("Tran Thi B");
        //st = new Student("Le Van C");
    }

   public  void test() {
     Student sv1 = new Student("Nguyen Van A");
     System.out.println("Before modify():"+sv1);
     modify(sv1);
     System.out.println("After modify():"+sv1);
   }
}
```

```
Before modify():Nguyen Van A
After modify():Tran Thi B
```

Passing a Reference Variable

- When you pass a reference variable in a method, you pass a copy of it and not the original reference variable
- The called method can change the object properties by using the passed reference
- Changing the object and the reference to the object are two different things, so note the following:
  - The original object can be changed in the called method by using the passed reference to the object
  - However, if the passed reference itself is changed in the called method, for example, set to null or reassigned to another object, it has no effect on the original reference variable in the calling method

Example -Passing a Reference Variable

```
public class MyCat {
    private String name;

    public MyCat(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Example -Passing a Reference Variable

```
public static void main(String[] args) {
    MyCat myCat = new MyCat("Kitty");
    System.out.println("Before call process: " + myCat.getName());
    process(myCat);
    System.out.println("After call process: " + myCat.getName());
}

public static void process(MyCat myCat) {
    myCat.setName("Doraemon");
}
```

```
public static void main(String[] args) {
    MyCat myCat = new MyCat("Kitty");
    System.out.println("Before call process: " + myCat.getName());
    process(myCat);
    System.out.println("After call process: " + myCat.getName());
}

public static void process(MyCat myCat) {
    myCat = new MyCat("Doraemon");
}
```

JavaBeans Naming Standard for Methods

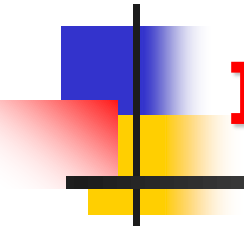
- Methods can be used to set the values of the class variables and to retrieve the values
- Methods written for these specific purposes are called **get** and **set** methods (also known as **getter** and **setter**)
- In special Java classes, called JavaBeans, the rules for naming the methods (including get and set) are enforced as a standard
- The private variables of a JavaBean called properties can only be accessed through its getter and setter methods

JavaBeans Naming Standard for Methods

- The rules
  - The naming convention for a property is: the **first letter of the first word in the name must be lowercase** and the **first letter of any subsequent word in the name must be uppercase**, e.g., myCow
  - The name of the getter method **begins with get** followed by the name of the property, with **the first letter of each word uppercased**
    - A getter method does not have any parameter and its return type matches the argument type
  - The setter method **begins with set** followed by the name of the property, with **the first letter of each word uppercased**
    - A setter method must have the **void** return type
  - The getter and setter methods must be public

JavaBeans Naming Standard for Methods

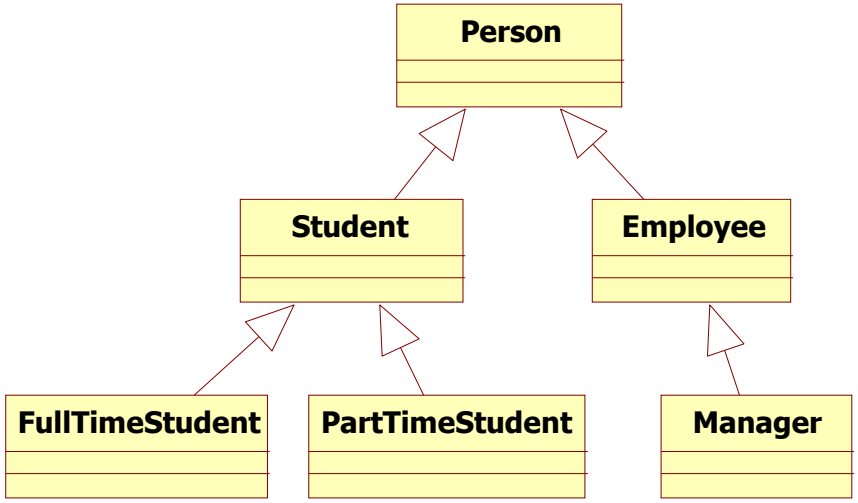
```
public class ScoreBean {
    private double meanScore;
    // getter method for property meanScore
    public double getMeanScore() {
       return meanScore;
    }
    // setter method to set the value of the property meanScore
    public void setMeanScore(double score) {
       meanScore = score;
    }
}
```



Inheritance & Polymorphism

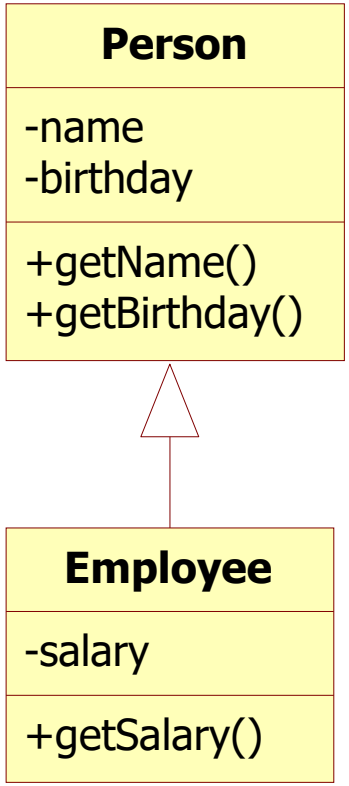
Inheritance

- Based on "is-a" relationship
- Subclass: more specialized, superclass: more general
- Subclass *is derived* or *inherits* from superclass
- In essence:
  - Objects in the same class have the same set of attributes (different values though) and operations
  - Objects of subclass have **all** members of superclass **plus** some more
  - Objects of a subclass can also be treated **as objects of its superclass**



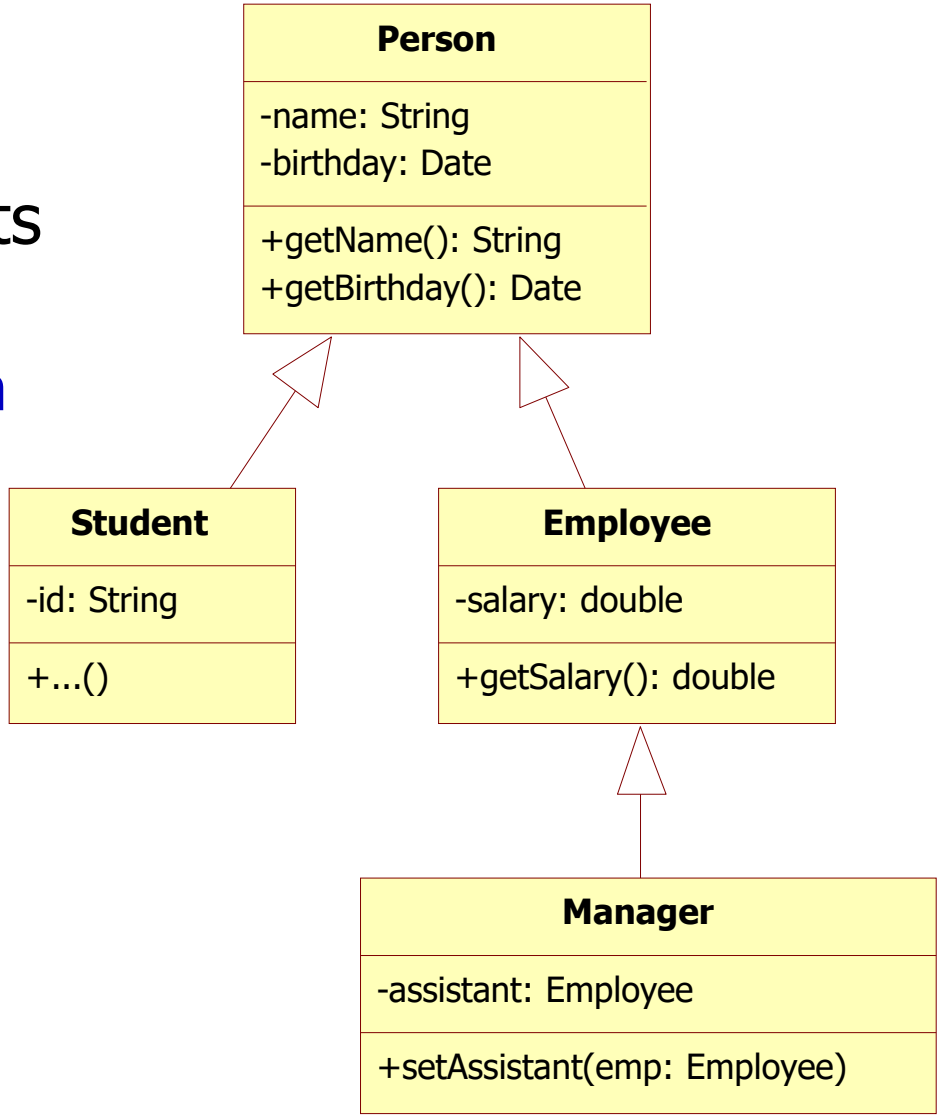
Inheritance

- An `Employee` "is a" `Person`,
  - apart from its own members, `salary` and `getSalary()`, it also has `name`, `birthday`, `getName()` without having to declare them
- `Employee` is the subclass (derived) of `Person`
- `Person` is the superclass (base) of `Employee`



Inheritance

- Inheritance tree can have many levels
  - A `Manager` object inherits what an `Employee` has, including what a `Person` has.



Defind inheritance in Java

- Using **extends** clause

```
public class Person {
    private String name;
    private Date birthday;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
}
```

```
class Employee extends Person {
    private double salary;
    public void setSalary(double salary) {
        this.salary = salary;
    }
}
```

New attributes/operations

```
//application code
...
Employee e = new Employee();
e.setName("John");
System.out.print(e.getName());
e.setSalary(3.0);
```

call to Person's **`setName()`** method

call to Employee's **`setSalary()`** method

Overide Method

- A subclass can redefine methods inherited from its superclass.
  - To specialise these methods to suit the new problem
- Objects of the subclass will work with the new version of the methods
  - Dynamic binding
- Superclass's methods of the same name can be reused by using the keyword **super**

Overriding

```
public class BankAccount {
    private float balance;
    public int getBalance() {
        return balance;
    }
}
```

```
public class InvestmentAccount extends BankAccount {
    private float cashAmount
    private float investmentAmount;
    public int getBalance() {
        return cashAmount + investmentAmount;
    }
}
```

More Example

- Subclass's version calls superclass's version then does something extra

```
public class Person {
  protected String name;
  protected Date birthday;
  public void display() {
    System.out.print (name + "," + birthday);
  }
...
}
```

Call method of the superclass from within the subclass. Keyword **super** is the reference to the superclass

```
public class Employee extends Person {
  ...
  public void display() {
    super.display();
    System.out.print ("," + salary);
  }
}
```

Method overriding - Rules

- New and old version must have the same prototype:
  - Same return type
  - Same argument type
- Private methods cannot be overriden
- Private members are hidden from subclass

Access control levels

Modifier	Accessible within			
	Same class	Same package	Subclasses	Universe
private	yes			
package (default)	yes	yes		
protected	**yes**	**yes**	**yes**	
public	yes	yes	yes	yes

protected access level

- protected members of a superclass are directly accessible from inside its subclasses.

```
public class Person {
    protected String name;
    protected Date birthday;
    ...
}
```

Subclass can directly access superclass's **protected** members

```
public class Employee extends Person {
    ...
    public String toString() {
        String s;
        s = name + "," + birthday; //no error.
        s += "," + salary;
        return s;
    }
}
```

In the same package

- Default access level is "**package**", which means those with "**package**" access level can be accessed directly from within the same package

```
package people;
public class Employee extends Person {
  ...
  public String toString() {
    String s;
    s = name + "," + birthday; //no error.
    s += "," + salary;
    return s;
  }
}
```

```
package people;
public class Person {
  String name;
  Date birthday;
  ...
```

In different packages

- Members with "package" access level **cannot** be accessed directly by subclasses from outside the package

```
package other;

import people.Person;
public class Employee extends Person {
  ...
  public String toString() {
    String s;
    s = name + "," + birthday; // error.
    s += "," + salary;
    return s;
  }
}
```

```
package people;
public class Person {
  String name;
  Date birthday;
  ...
```

In different packages

- Members with "**protected**" access level **can** be accessed directly by subclasses from outside the package

```
package people;
public class Person {
  protected String name;
  protected Date birthday;
  ...
```

```
package other;

import people.Person;
public class Employee extends Person {
  ...
  public String toString() {
    String s;
    s = name + "," + birthday; // OK
    s += "," + salary;
    return s;
  }
}
```

Constructor of subclass

- Subclass inherits all attributes/ methods of superclass
  - Subclass must initialize inherited members
- But, **constructors are NOT inherited**
  - syntactically, they have different names
- Two ways to call constructors of the baseclass

  1. (Implicit- ngầm định) use default constructors
  2. Explicit (tường minh) calls to constructors of the baseclass

Call default constructor

```
class Shape {
   protected int x, y;
   public Shape() {}
   public Shape(int x, int y) {
      this.x = x;
      this.y = y;
   }
}
class Circle extends Shape {
   protected int radius;
   public Circle() {}
}
```

The compiler supplies an implicit **`super()`** constructor for all constructors.

Default constructor **`Shape()`** is called

```
// client code
Shape p = new Shape(10, 10);
Circle c1 = new Circle();
Circle c2 = new Circle(10, 10); // error
```

Cannot found **`Circle(int, int)`**.
`Shape(int, int)` is not inherited

Calling constructors of baseclass

- The initializing superclass ' attributes should be carried out by baseclass' constructors
  - Why?
- Superclass' constructors can be called using reference **`super`**
  - Superclass' constructors must run first
    `super(argument list);`
  - If superclass has no default constructor then its constructor must be called explicitly

Calling constructors of baseclass

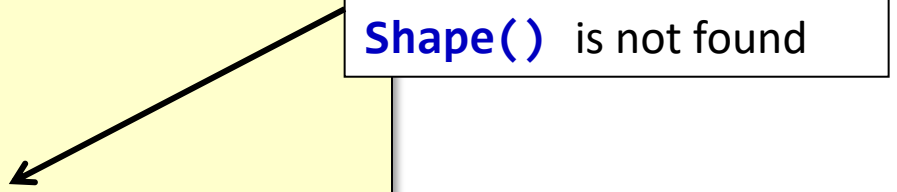
```
class Shape {
    protected int x, y;
    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Circle extends Shape {
    protected int radius;
    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }
}
```

Explicit calls to **Shape(int, int)**

```
// client code
Shape p = new Shape(10, 10);
Circle c2 = new Circle(10, 10, 5); // OK
```

Calling constructors of baseclass

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

class Circle extends Shape {
    protected int radius;
    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
}
```



toString() method

- Inherits from Object class

```
class Shape {
   protected int x, y;
   public String toString() {
      return "<" + x + "," + y + ">";
   }
}
class Circle extends Point {
   protected int radius;
   public String toString() {
      return super.toString() + "," + radius;
}
}
```

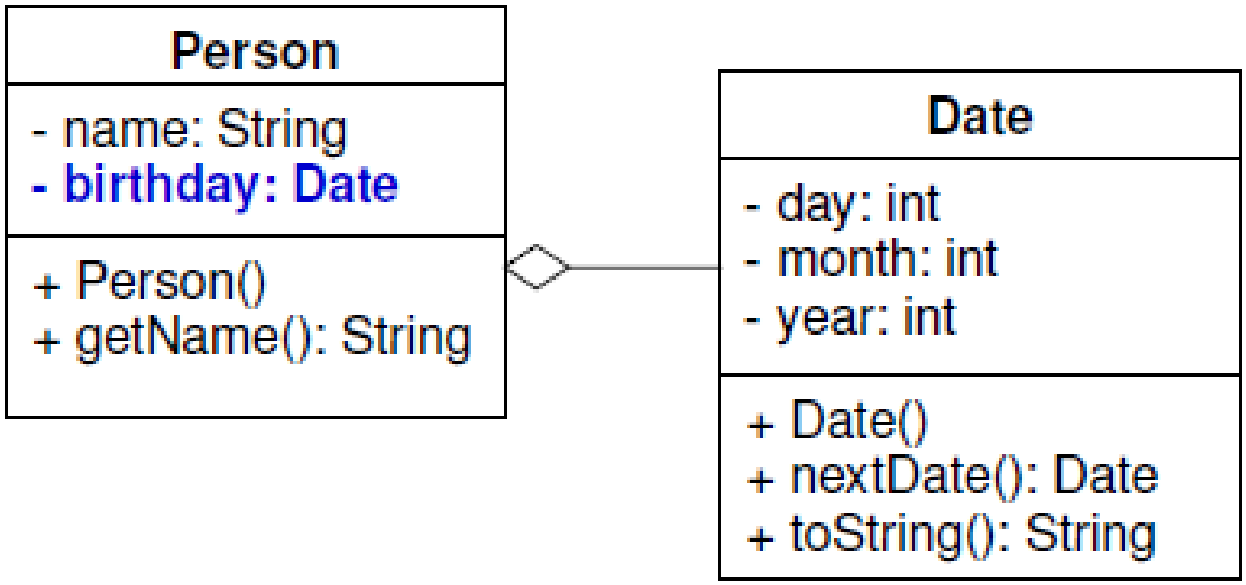
Overriding Object's **toString()**
New versions are used in
System.out.println()

```
// client code
Circle c = new Circle();
System.out.println(c);
```

Reusing Classes

- Object classes with similar or related attributes and behaviour
  - Person, Student, Manager,...
- Code reuse
  - Composition – "has-a" relationship
    - the new class is composed of objects of existing classes.
    - reuse the functionality of the existing class, not its form
  - Inheritance – "is-a" relationship
    - create a new class as a *type of* an existing class
    - new class absorbs the existing class's members and extends them with new or modified capabilities

Reusing classes - composition



- A class can have references to objects of other classes as members.
- This is called **composition** and is sometimes referred to as a **has-a relationship**.

Example

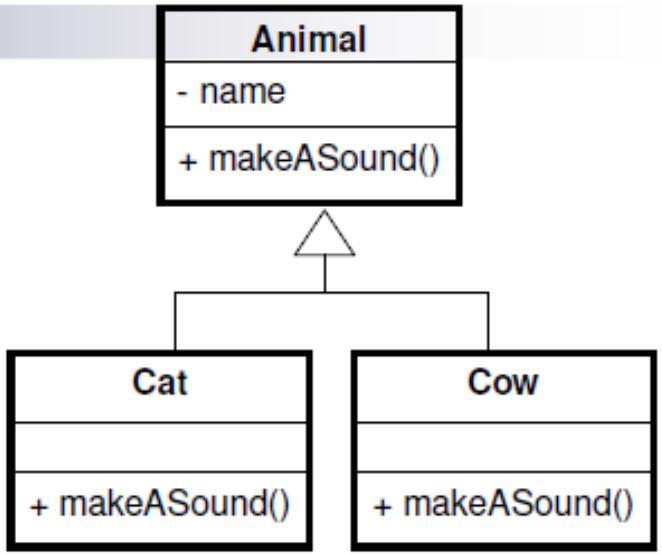
```
public class Student {
   private String name;
   private Date birthday;

   public Student(String name,
                  Date birthday) {
      this.name = name;
      this.birthday = birthday;
   }
}
```

```
public class Date {
   private int day;
   private int month;
   private int year;

   public Date(int day,
               int month,
               int year) {
      this.day = day;
      this.month = month;
      this.year = year;
   }
}
```

What is polymorphism?

- Polymorphism: *exist in many forms*
  - Polymorphism in programming
  - Function polymorphism: same name, different arguments
- Object polymorphism:
  - An object can be treated in different ways
    - A Manager object can be seen as an Employee object as well
  - Different objects interpret the same message differently
    - How do kangaroos and frogs "**jump**"?

```
class Animal {
  ...
  public void makeASound() {
    System.out.print ("Uh oh!");
  }
}
class Cat extends Animal {
  ...
  public void makeASound() {
    System.out.print ("Meow...");
  }
}
class Cow extends Animal {
  ...
  public void makeASound() {
    System.out.print ("Moo...");
  }
}
```



Polymorphism:
The same message "makeASound" is interpreted differently

Meow... Moo...

```
Animal myPets[] = new Animal[2];
myPets[0] = new Cat("tom");
myPets[1] = new Cow("mini");
for ( int i = 0; i < myPets.length; i++ ) {
  myPets[i].makeASound();
}
```

```
class Animal {
  ...
  public void makeASound() {
    System.out.print ("Uh oh!");
  }
  public void introduce() {
    makeASound();
    System.out.println(" I'm " + name);
  }
}
class Cat extends Animal {
  ...
  public void makeASound() {
    System.out.print("Meow...");
  }
}
class Cow extends Animal {
  ...
  public void makeASound() {
    System.out.print("Moo...");
  }
}
```

Polymorphism: The same message "makeASound" is interpreted differently

```
Animal pet1 = new Cat("Tom Cat");
Animal pet2 = new Cow("Mini Cow");
pet1.introduce();
pet2.introduce();
```

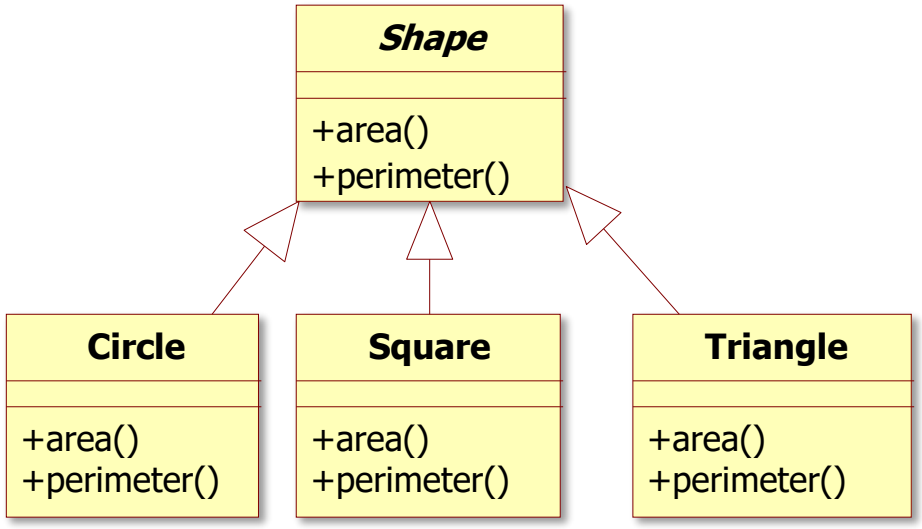
Meow... I'm Tom Cat
Moo... I'm Mini Cow

Dynamic & static binding

- Method binding: connect a method call to a method body
- Static/early binding: performed by compiler/linker before the program is run.
  - The only option of procedural languages.
- Dynamic/late binding: performed during run-time
  - Java uses late binding, except for static, final, and private methods.
    - private methods are implicitly final.

Abstract class

- Sometimes we don't want objects of a base class to be created

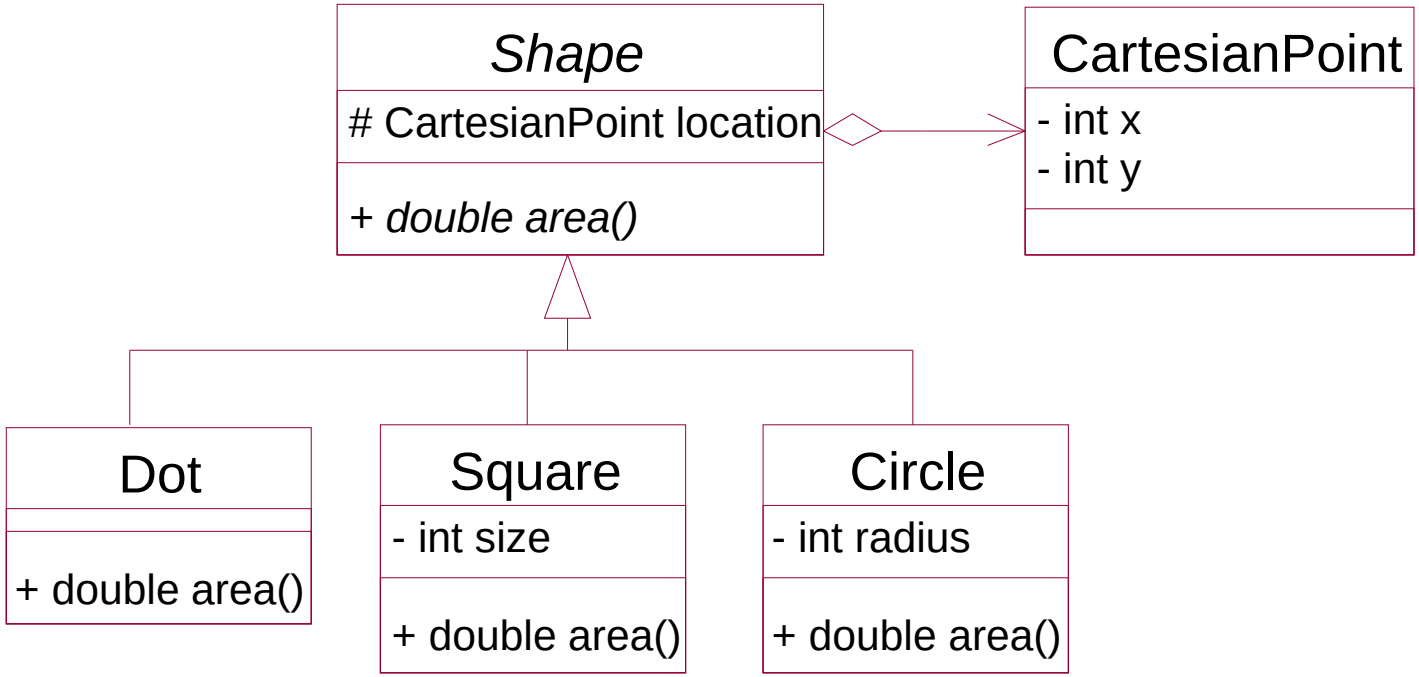


- Examples:
  - Animal, Cat, Cow, Dog,…
    - An Animal object makes no sense
    - What sort of sound does it make?
  - Shape, Point, Rectangle, Triangle, Circle
    - What does a generic Shape look like?
    - How to draw it?
- Solution: make it an abstract base class

Abstract Classes

- Abstract classes cannot be instantiated – they are intended to be a superclass for other classes
- abstract methods have no implementation
- If a class has one or more abstract methods, it is abstract, and must be declared so
- Concrete classes have full implementations and can be instantiated

Example


Shape
CartesianPoint location
+ double area()
CartesianPoint
- int x
- int y
Dot
+ double area()
Square
- int size
+ double area()
Circle
- int radius
+ double area()

```java
public class CartesianPoint {
    private int x;
    private int y;
    public CartesianPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

```java
public abstract class Shape {
    protected CartesianPoint location;
    public Shape(CartesianPoint location) {
        this.location = location;
    }
    public abstract double area();
    public abstract double perimeter();
    public abstract void draw();
}
```

```
public class Circle extends Shape {
   private int radius;

   public Circle(CartesianPoint location, int radius) {
      super(location);
      this.radius = radius;
   }

   public double area() {
      return Math.PI * radius * radius;
   }

   public double perimeter() {
      return 2 * Math.PI * this.radius;
   }

   public void draw() {
      System.out.println("Draw circle at ("+ x + ","
                         + y + ")");
   }
```

BÀI TẬP

- Quản lý các đối tượng trong một học viện:
  - Nhân viên quản lý (mã nv, tên nv, trình độ, chuyên môn, lương cơ bản, phụ cấp chức vụ)

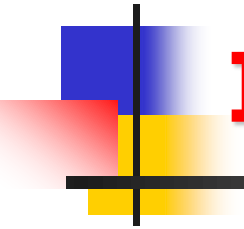
    Lương = lương cơ bản + phụ cấp chức vụ
  - Nhân viên nghiên cứu (mã nv, tên nv, trình độ, lương cơ bản, phụ cấp độc hại)

    Lương = lương cơ bản + phụ cấp đọc hại
  - Nhân viên phục vụ: (mã nv, tên nv, trình độ, lương cơ bản)

    Lương = lương cơ bản

BÀI TẬP

- Xây dựng lớp **HocVien** (học viện) để quản lý danh sách nhân viên của học viện, và thực hiện các yêu cầu sau:
  - In ra danh sách các nhân viên
  - Tính tổng lương của tất cả các nhân viên có trong học viện
  - Tìm nhân viên có mức lương thấp nhất, và in ra thông tin của nhân viên này.



Interface

Java interfaces

- Java does not support multiple inheritance
  - This is often problematic
    - What if we want an object to be multiple things?
- Interfaces
  - A special type of class which
    - Defines a set of method prototypes
    - Does not provide the implementation for the prototypes
    - Can also define final constants

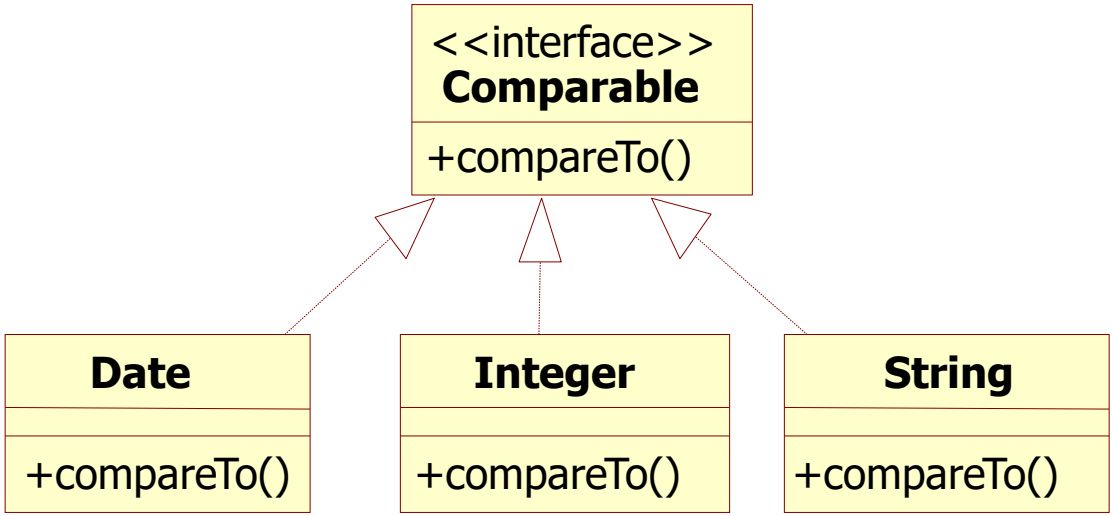
Interfaces

- Declared types in Java are either classes or interfaces
- Interfaces represent a promise of support services to the objects which implement the interface – a "contract"

```
public interface File {
    public void open(String name);
    public void close();
}
```

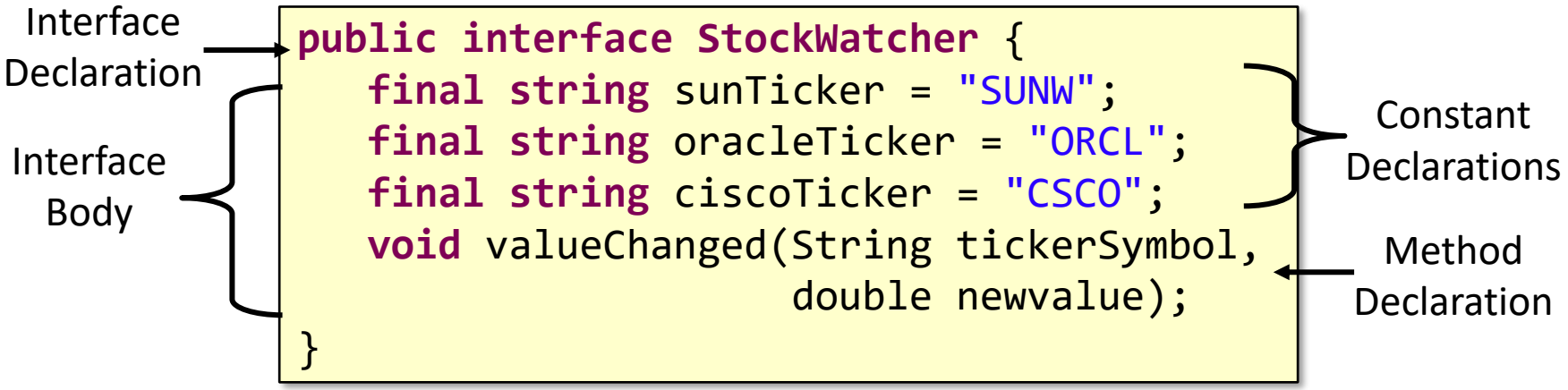
Protocols

- An interface defines a protocol (a set of methods)
  - If a class implements a given interface, then that class implements that protocol.
- An interface can impose a common protocol on a group of classes that are not related by inheritance
  - In the chain of classes related by inheritance, the common protocol is imposed through subclassing



Declaring an Interface

- Interface definition has two components
  - Interface declaration
  - Interface body
- The interface declaration declares various attributes about the interface
  - Name
  - Whether it extends other interfaces



Implementing Interface Methods

- Methods declared in an interface are implemented in the classes which support that interface

```
public interface File {
    public void open(String name);
    public void close();
}
```

```
public class TextFile implements File {
    public void open(String name) {
        // implementation of open method
    }
    public void close() {
        // implementation of close method
    }
}
```

Syntax

- In a class declaration, the naming of the superclass precedes any interfaces supported by the class:

```
public class Directory extends Secure implements File {
    ...
}
```

- Multiple Interfaces:
  - If a class implements multiple interfaces, the interfaces are all listed, separated by commas

```
public class Directory implements File, Secure {
    ...
}
```

Implementing an Interface

- A class which implements an interface must implement every method defined by that interface
- If one or more methods is not implemented, Java will generate a compiler error
- Subclasses automatically implement all interfaces that their superclass implements

Typing and Interfaces

- A variable's type can be an interface

- Stipulations (quy định)
  - Only objects whose class implements that interface can be bound to that variable
  - Only messages defined by the interface can be used
  - Interfaces cannot appear in a **`new`** expression

```
File r = new File(); // Error
File f = new TextFile(); // OK!
```

Subinterfaces

- Interfaces can be extended
  - Interface hierarchy is independent of the class hierarchy
  - The interface which extends another interface inherits all of its method declarations

```
interface File {
   public void open(String name);
   public void close();
}
```

```
interface ReadableFile extends File {
   public byte readByte();
}
```

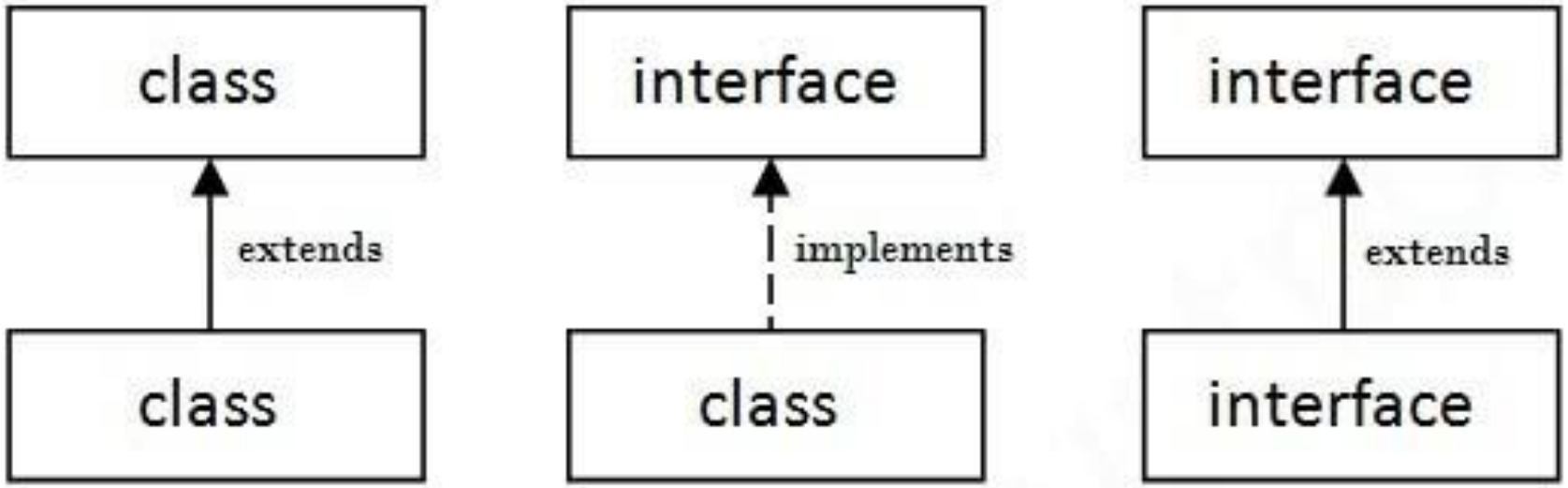
```
interface WritableFile extends File {
   public void writeByte(byte b);
}
```

```
interface ReadWriteFile
                  extends ReadableFile, WritableFile {
   public void seek(int position);
}
```

Using Interfaces

- Using interfaces allows
  - Cross-hierarchy polymorphism
  - Access to methods in separate class trees
  - Substitution of an object for another object
- Classes that implement the same interface understand the same messages
  - Regardless of their location in the class hierarchy

Using Interfaces



More Advantages of Using Interfaces

- Allows more control over how objects are used
- The programmer can define a method's parameters as interfaces
  - This restricts the use of those parameters
  - The programmer knows which message the objects will respond to
- Improves reusability of code

Naming Conventions for Interfaces

- Make "able" interfaces
  - `Cloneable, Serializable, ...`
- Name interfaces using proper nouns and provide "Impl" implementation of your interfaces
  - `Bank, BankImpl, BankAccount, BankAccountImpl`
  - With this convention, the interface typically contains a definition for all (or most) of the implementation class' public methods
- Prefix interface names with "I" and use proper nouns for your classes
  - `IBank, Bank, IBankAccount`

The Interface Comparable

```
interface Comparable{
// compare this object with the given object
// produce negative result if this is 'smaller' than the given object
// produce zero if this object is 'the same' as the given object
// produce positive result if this is 'greater' than the given object
  int compareTo(Object obj);
}
```

Case study: Comparable Interface

```
public class Employee implements Comparable{
    private String name;
    private double salary;

    public Employee(String n, double s) {
        name = n;
        salary = s;
    }

    @Override
    public int compareTo(Object o) {
        Employee emp = (Employee)o;
        return this.getName().compareTo(emp.getName());
    }
}
```

Sort Array of Comparable Objects

```
public class EmployeeSortTest {
  public static void main(String[] args) {
    Employee[] staffs = new Employee[3];

    staffs[0] = new Employee("Harry Hacker", 35000);
    staffs[1] = new Employee("Carl Cracker", 75000);
    staffs[2] = new Employee("Tony Tester", 38000);

    Arrays.sort(staffs);

    // print out information about all Employee
    for (Employee e : staffs)
      System.out.println("Name = " + e.getName()
          + ", Salary = " + e.getSalary());
}
```

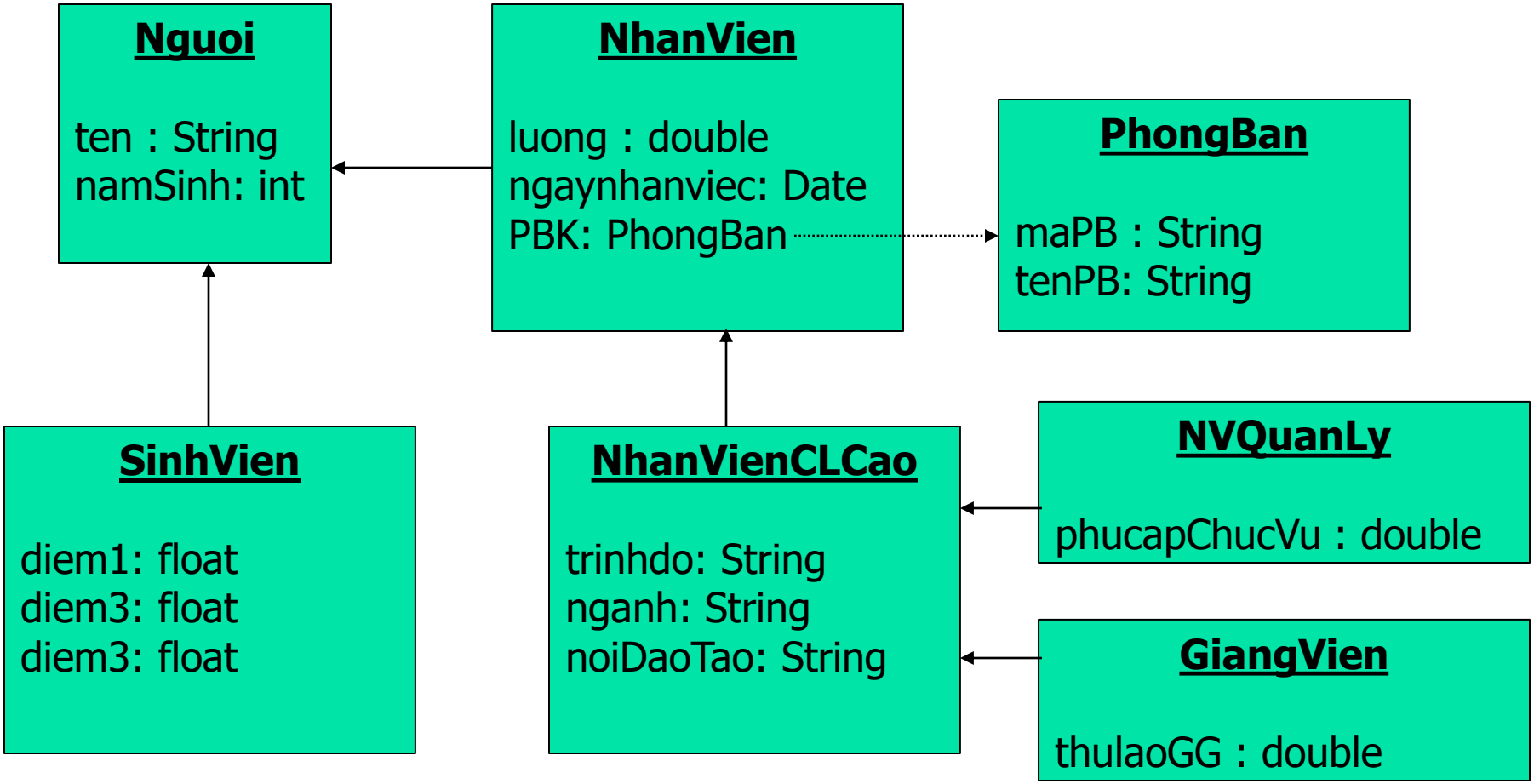
Sort Array using Comparator Interface

```
public class Employee {
   private String name;
   private double salary;
   public Employee(String n, double s) {
      name = n;   salary = s;
   }
}
```

```
public class EmployeeSortTest {
  public static void main(String[] args) {
    Employee[] staffs = new Employee[3];
    staffs[0] = new Employee("Harry Hacker", 35000);
    staffs[1] = new Employee("Carl Cracker", 75000);
    staffs[2] = new Employee("Tony Tester", 38000);
    Arrays.sort(staffs, new Comparator<Employee>() {
       public int compare(Employee o1, Employee o2) {
          return o1.getName().compareTo(o2.getName());
       }
    });
    // print out information about all Employee
    for (Employee e : staffs)
      System.out.println("Name = " + e.getName()
          + ", Salary = " + e.getSalary());
}
```

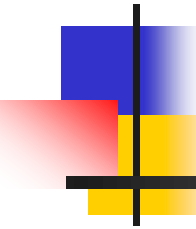
BÀI TẬP

- Cho sơ đồ cây phân cấp như sau:



BÀI TẬP

- Xây dựng các lớp theo sơ đồ cây phân cấp thừa kế, Xây dựng lớp **TruongDaiHoc** để thực hiện các yêu cầu sau
  - In ra danh sách các sinh viên
  - In ra danh sách các nhân viên
  - In ra danh sách lương của nhân viên
  - Tính lương của tất cả các nhân viên
    - Với lương nhân viên quản lý = lương + phụ cấp chức vụ
    - Với lương của giảng viên = lương + thù lao giảng dạy.
  - Tìm nhân viên có tiền lương cao nhất.
  - Viết phương thức kiểm tra Người có phải là sinh viên hay không?



Inner Class

Introduction

- Inner classes let you define one class within another
- They provide a type of scoping for your classes since you can make one class a member of another class
- Just as classes have member variables and methods, a class can also have member classes

Inner Class

- Sometimes, though, you find yourself designing a class where you discover you need behavior that belongs in a separate, specialized class, but also needs to be intimately tied to the class you're designing
- Event handlers are perhaps the best example of this
  - A Chat client specific methods in the ChatClient class, and put the event-handling code in a separate event-handling class

Inner Class (cont.)

- One of the key benefits of an inner class is the "special relationship" an inner class instance shares with an instance of the outer class
- That "special relationship" gives code in the inner class access to members of the enclosing (outer) class, as if the inner class were part of the outer class
  - An inner class instance has access to all members of the outer class, even those marked private

Regular Inner Class

- A regular inner class can't have static declarations of any kind
- The only way you can access the inner class is through a live instance of the outer class
- Declare inner class

```
class MyOuter {
    class MyInner { }
}
```

- When compile MyOuter, It creates two classes MyOuter.class and MyOuter$MyInner.class

Regular Inner Class (cont.)

```
class MyOuter {
   private int x = 7;
   // inner class definition
   class MyInner {
       public void seeOuter() {
          System.out.println("Outer x is " + x);
       }
   } // close inner class definition
} // close outer class
```

Anonymous Inner Classes

- Inner classes declared without any class name are called anonymous
- You can even define anonymous classes within an argument to a method

```
class Popcorn {
    public void pop() {
       System.out.println("popcorn");
    }
}
class Food {
    Popcorn p = new Popcorn() {
        public void pop() {
            System.out.println("anonymous popcorn");
        }
    };
}
```

declare a new class which has no name, but which is a subclass of Popcorn

Overriding the pop() method

End of the anonymous class definition

Anonymous Inner Classes with Interface

```
interface Cookable {
    public void cook();
}
class Food {
    Cookable c = new Cookable() {
        public void cook() {
          System.out.println("anonymous cookable implementer");
        }
    };
}
```

Anonymous Inner Classes as Arguments

```
class MyWonderfulClass {
   void go() {
      Bar b = new Bar();
      b.doStuff(new Foo() {
         public void foof() {
            System.out.println("foofy");
         } // end foof method
      }); // end inner class def, arg, and end statement
   } // end go()
} // end class

interface Foo {
   void foof();
}
class Bar {
   void doStuff(Foo f) { }
}
```

Conversion of Data Types

- Two kinds of Conversion of Data Types
  - Conversion of Primitive Data Types
  - Conversion of Reference Data Types
- Conversion of Data Types
  - **Implicit type conversion**: The programmer does not make any attempt to convert the type
  - **Explicit type conversion**: Conversion is initiated by the programmer by making an explicit request for conversion (type casting)

Conversion of Data Types

- Assignment Conversion

<sourceType> s = new <sourceType>();

<targetType> t = s; // Implicit conversion of <sourceType> to <targetType>

- Method Call Conversion
- Arithmetic Conversion

Implicit Primitive Type Conversion

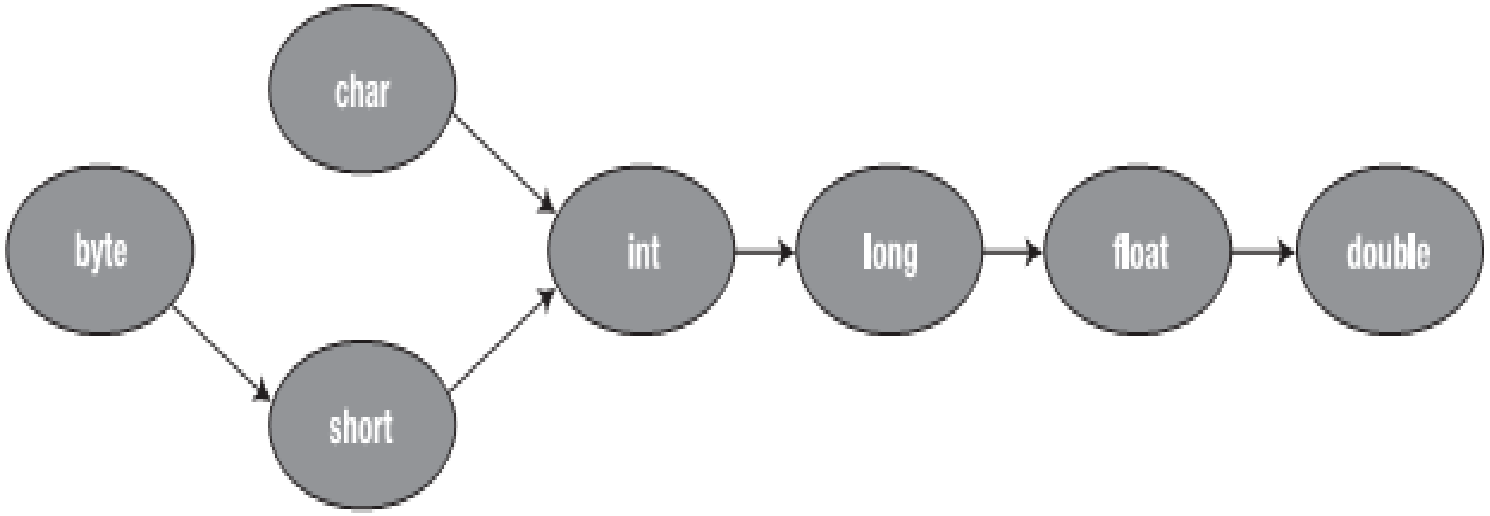
- Two general rules for implicit primitive type conversion are the following:
  - There is no conversion between boolean and non-boolean types.
  - A non-boolean type can be converted into another non-boolean type only if the conversion is not narrowing—that is, the size of the target type is greater than or equal to the size of the source type.

Implicit Primitive Type Conversion

```
public class ConversionToWider{
    public static void main(String[] args) {
        int i = 15;
        short s = 10;
        s= i;
        System.out.println("Value of i: "  + i );
    }
}
```

Error: Illegal conversion

Implicit Primitive Type Conversion



Primitive Data Type	Allowed Implicit Conversion To
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Explicit Primitive Type Conversion

- In a narrowing conversion, casting is mandatory
- However, casting to a narrower type runs the risk of losing information and generating inaccurate results
- You cannot cast a boolean to a non-boolean type.
- You cannot cast a non-boolean type to a boolean type.