COSC 123
Computer Creativity

Java Classes

Dr. Ramon Lawrence
University of British Columbia Okanagan
ramon.lawrence@ubc.ca

Key Points

1) Define classes, objects, methods, properties (instance variables), and parameters in Java.

2) Inheritance derives new classes from existing ones. A subclass inherits all methods and variables from its superclass.

3) Create objects from classes using `new`.

4) Explain the difference between an object and an object reference.

5) List the types of variables (instance, static, local, parameter) and explain how the type affects their scope and lifetime.

Java Object-Oriented Terminology

An ***object*** is an instance of a class that has its own properties and methods. Properties and methods define what the object is and what it can do. *Each object has its own area in memory.*

A ***class*** is a generic template (blueprint) for creating an object. All objects of a class have the same methods and properties (although the property values can be different).

A ***property*** (or ***instance variable***) is an attribute of an object.

A ***method*** is a set of statements that performs an action. *A method works on an implicit object and may have parameters.*

A ***parameter*** is data passed into a method for it to use.

Class Definition

To define a class:

- use the keyword **`class`** and provide a name for your class
- enclose the contents of your class in brackets "**{**", "**}**"
- define any properties (*instance variables*) for your class
- define any methods in your class

Example:

```
class classname
{  classname methods
   classname variables
}
```


Variable Definition

To define a variable of a class:

- define the variable as either public or private (access specifier)
- provide the variable type and name as usual

Syntax:

```
class classname
{  accessSpecifier variableType variableName;
   ...
}
```

Example:

```
class MyClass
{  public int num;
   private String st;
   private double value;
}
```


Method Definition

To define a method of a class:

- define the method as either public or private (access specifier)
- provide the method return type, name, and parameters
  - ⇨Each parameter has a type and a name.
  - ⇨A return type of void means return nothing.

Syntax:

```
class classname
{  accessSpec retType methodName(par1, par2, …, parN)
   { method implementation}
}
```

Example:

```
class TestClass
{  public int count(int n)        { return n+1; }
   private void doNothing()       { }
   public String addS(String st) { return st+"S"; }
}
```

Method Definition Parameters

A method may use parameters to perform its operations.

- Each parameter has a type and a name.
- Parameters are separated by commas.
- Parameters can be changed by the method, but their value will not be changed for the caller.

Method Definition Return Types

Use the **`return`** statement to return a method value. Syntax:

```
return expression;  OR

return;
```

Example:

```
class TestClass
{  public int retTest(int n)
   {  if (n == 0)
         return 1;
      else
         return n*2+1;
   }
   public void retNothing(String st)
   {  if (st.equals(""))
         return;
      ...
   }
}
```


Your First Java Program (again)

```
public class HelloWorld
{  public static void main(String[] args)
   {  System.out.println("Hello, World!");
   }
}
```

The first line creates a ***public class*** called `HelloWorld` that is the main class of your program and the name of the Java file.

Class `HelloWorld` contains a method `main` that is `public`.

- Since class `HelloWorld` is the public class for this file, it is the class that must contain the `main` method.
- `main` is a method called with one parameter (`String[] args`).
- `main` is a special method because it is automatically called when you run your program.

Class Example `BankAccount` Class

```
public class BankAccount
{
   private double balance;

   public void deposit(double amount)
   {  balance = balance + amount; }

   public void withdraw(double amount)
   {  balance = balance - amount; }

   public double getBalance()
   {  return balance; }
}
```

The `BankAccount` class is used for describing bank accounts.

- The methods defined in the `BankAccount` class are `deposit`, `withdraw`, and `getBalance`.
- The current balance in the account is `private`, so it can only be changed by calling the methods.

Practice Questions

1) Implement a class `Employee`:

- An employee has a name (String) and a salary (`double`).
- Write methods to get/set the name and salary.

2) Implement a class `Purse`:

- A purse holds coins (toonies, loonies, and quarters only).
- Write methods to get/set the number of coins in the purse.
- Write a method called `getValue()` which returns the value of all coins in the purse.

Inheritance Overview

Inheritance is a mechanism for enhancing and extending existing, working classes.

- ⇨ In real life, you inherit some of the properties from your parents when you are born. However, you also have unique properties specific to you.
- ⇨ In Java, a class that extends another class inherits some of its properties (methods, instance variables) and can also define properties of its own.

`extends` is the key word used to indicate when one class is related to another by inheritance.

Syntax: `class subclass` **`extends`** `superclass`

- The ***superclass*** is the existing, parent class.
- The ***subclass*** is the new class which contains the functionality of the superclass plus new variables and methods.
- A subclass may only inherit from ***one*** superclass.

Why use inheritance?

The biggest reason for using inheritance is to re-use code.

- Once a class has been created to perform a certain function it can be re-used in other programs.
- Further, using inheritance the class can be extended to tackle new, more complex problems without having to re-implement the part of the class that already works.

The alternative is copy and paste which is bad, especially when the code changes.

What is inherited?

When a subclass inherits (or extends) a superclass:

Instance variable inheritance:

- All instance variables of the superclass are inherited by the subclass.
  - ⇨ However, if a variable is **`private`**, it can only be accessed using methods defined by the superclass.

Method inheritance:

- All superclass methods are inherited by the subclass, but they may be ***overridden***.

Inheritance Example

Consider the `BankAccount` class that we created to model bank account objects.

- A bank account has an account number and a balance.

How about if we want to create a special kind of bank account called a `SavingsAccount`?

- A savings account is a special bank account because it also pays interest at a given interest rate.
- Instead of programming the entire `SavingsAccount` class and duplicating features already in the `BankAccount` class, we can extend the `BankAccount` class and inherit its properties when we create a `SavingsAccount`.

BankAccount Code

```
public class BankAccount
{  public void deposit(double amount)
   {  balance = balance + amount; }
   public void withdraw(double amount)
   {  balance = balance - amount; }
   public double getBalance()
   {  return balance; }
   public int getAccount()
   {  return accountNum; }
   public int getLastAccount()
   {  return lastAccountNum; }
   public BankAccount()
   {  this(0); }
   public BankAccount(double b)
   {  balance = b; lastAccountNum++;
      accountNum = lastAccountNum;
   }
   private double balance;
   private int accountNum;
   private static int lastAccountNum = 0; // Static
}
```


SavingsAccount Code

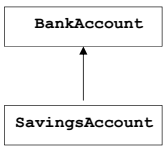
```
public class SavingsAccount extends BankAccount
{
   public void addInterest()
   {  deposit(getBalance()*rate/100); }

   public SavingsAccount()
   {  this(0); }
   public SavingsAccount(double r)
   {  rate = r;   }

   private double rate; // Interest rate paid
}

Notes:
1) Inherited variables: balance,accountNum,lastaccountNum
2) Inherited methods: deposit, withdraw, getAccount,
                      getLastAccount
3) Inherited variables are private in BankAccount, so we
    cannot access them directly.
   (Use deposit and getBalance methods.)
```


Class Diagrams

Class diagrams display the relationship between related classes using a diagram.

- We will follow the Unified Modeling Language (UML) syntax.
  - ⇨ Each class has its own box.
  - ⇨ There is an arrow from a subclass to a superclass if that class extends the superclass.

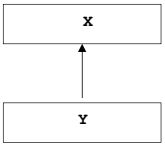
Example:




```
SavingsAccount
extends BankAccount
```

Superclass and Subclass

Question: Which class is the superclass?



A) X
B) Y

Inheritance

Question: Which statement is true?

A) A subclass can access all variables it inherited from the superclass.

B) A subclass can declare an instance variable with the same name as an instance variable in its superclass.

C) A class can have more than one superclass.

Access Specifiers Public and Private

One of the features of object-oriented programming is that not all parts of a program have access to all the data and methods.

Each class, method, and *instance* variable has one of the four ***access specifiers*** to indicate which other objects and methods in your program have access to it. Four types:

- `public` – Accessible by all code (everyone, the public)
- `private` – Only accessible by methods in the class.
- `protected` – Only accessible by methods in the class or classes derived from this class by inheritance.
- default – If nothing is specified, assume package access where all methods in same package (directory) can access it.

Public and Private Examples

```
public class MyClass
{  public void setValue(int n)
   {  num = n; }          // setValue() is a public method

   private void show()    // show() is a private method
   {  st = "Hello"; }

   private int num;       // num is a private variable
   public String st;      // st is a public variable
   double d;              // d has package access
}
Summary:
1) Method setValue() is public, so it can be accessed
from anywhere in the program.
2) Method show() is private so only another method in the
class MyClass can access it.
3) Variable num is private, only methods in MyClass can
access it.
4) st is public. It is accessible anywhere in the program.
5) d has default (package) access.  Any method in a file
 in the same package (directory) can access it.
```


Access Specifier Rules

There is one special rule in Java that you must follow:

- There can be only one public class per file, and the name of that class has to be the same as the name of the file.

There are also some common programming rules which you will use in this course:

- Always state if a class/variable/method is `public` or `private`.
- Variables in an object are almost always `private`.
  - ⇨ Other objects/methods do not have access to the data directly.
- Most methods of an object are `public`.
  - ⇨ These methods allow other objects/methods to see/manipulate the data.
- Class names should begin with a capital letter.
- Method and variable names should begin with a small letter.

Inheritance Question

1) Create a `CheckingAccount` class which inherits from `BankAccount`. The `CheckingAccount` class:

- inherits `getBalance()` from `BankAccount`
- overrides `deposit()` and `withdraw()` from `BankAccount`, so it can keep track of the number of transactions (`transactionCount`)
- defines a method `deductFees()` which withdraws $1 for each transaction (`transactionCount`) then resets the # of transactions

Inheritance Questions (2)

2) Create:

- 1) A superclass `Pet`. A `Pet` contains:
  - ⇨ a `name` and methods to get/set its `name`
- 2) A subclass `Cat` of `Pet`. A `Cat` contains:
  - ⇨ a boolean variable `hasClaws` which is true if the cat has claws
  - ⇨ define methods to get/set the `hasClaws` instance variable
- 3) A subclass `Dog` of `Pet` that contains:
  - ⇨ an integer variable `numTricks` that stores the number of tricks the dog can perform
  - ⇨ define methods to get/set the value of `numTricks`

Creating and Using Objects

A class is just a blue-print for creating objects.

- By itself, a class performs no work or stores no data.

For a class to be useful, we must create objects of the class.

- Each object created is called an ***object instance***.

To create an object, we use the **`new`** method.

When an object is created using the `new` method:

- Java allocates space for the object in memory.
- The ***constructor*** for the object is called to initialize its contents.
- Java returns a pointer to where the object is stored in memory which we will call an ***object reference***.

Constructors

A ***constructor*** is a method that is called when the object is first created and initializes the variables of an object.

- If you do not supply a constructor for a class, Java supplies a ***default constructor*** which has no parameters.
- You may define your own constructors for your objects to guarantee that an object has the correct initial values.
  - ⇨ A constructor may have parameters like any other method.

Syntax and Example:

```
class classname                // (Syntax)
{  classname() {}              // Default constructor
   classname(par1, par2, …, parN) {} //Parameters
}
class MyClass                   //(Example)
{  MyClass()       { num = 0; } // Default constructor
   MyClass(int n) { num = n; } // Parameters
   private int num;             // Variable initialized
}
```


Bank Account Example Revisited

```
public class BankAccount
{  public void deposit(double amount)
   {  balance = balance + amount; }
   public void withdraw(double amount)
   {  balance = balance - amount; }
   public double getBalance()
   {  return balance; }

   public BankAccount()          { balance = 0; }
   public BankAccount(double b)  { balance = b; }

   private double balance;
}
```

The `BankAccount` class now defines two constructors:

- Default constructor initializes balance to 0.
- Constructor with parameter initializes balance to a given value.

Creating Objects using `new`

Objects are created using the **`new`** method.

The new method allocates space for the object in memory, calls the appropriate object constructor, and returns an ***object reference*** to be stored in an object reference variable.

Example:

```
BankAccount checking = new BankAccount();
// Creates a BankAccount object referenced by checking
BankAccount savings = new BankAccount();
// Creates a BankAccount object referenced by savings

BankAccount mySavings;  // Declares object reference

mySavings = new BankAccount(); // Creates object
```


Object References

It is important to realize the difference between an object and an object reference.

When you declare an object variable in Java, you are actually declaring an object reference to that particular object type.

- Until you create an object using the `new` method, there is no object in memory which is pointed to by the object reference.

An object is the physical memory representation of the data.

- An object has a location in memory and a type (class).
  - ⇨ Each object has its own data values.

Changing Object References

Object references are pointers to objects in memory that can be assigned to the same value as another reference using '=' or assigned to **null** (which means they refer to nothing).

Example:

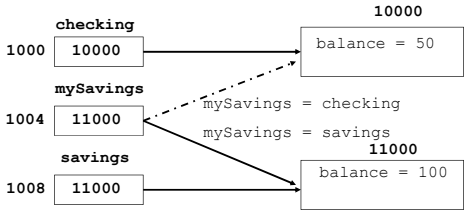
```
BankAccount checking = new BankAccount(50);
// Creates a BankAccount object referenced by checking
BankAccount savings = new BankAccount(100);
// Creates a BankAccount object referenced by savings
BankAccount mySavings;  // Declares object reference

mySavings = savings;    // mySavings points to savings
System.out.println(mySavings.getBalance()); // 100
mySavings = checking;   // mySavings points to checking
System.out.println(mySavings.getBalance()); // 50
```


Objects in Memory

Remember that each object has its own space in memory AND each object reference variable also has its own memory space.

Object references point to objects and can be changed.

Memory diagram based on previous example:



null Object References

Sometimes a programmer wants an object reference to point to nothing. To make an object reference refer to nothing, you assign it a value of **null**.

Example:

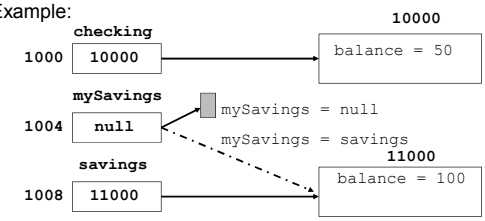
```
BankAccount checking = new BankAccount(50);
BankAccount savings = new BankAccount(100);
BankAccount mySavings;  // Declares object reference

mySavings = savings;    // mySavings points to savings
System.out.println(mySavings.getBalance()); // 100
mySavings = null;       // mySavings now points to null
System.out.println(mySavings.getBalance()); // Error!
```


null Object References Example

A null reference effectively stores the address of **0**. Since this is not a valid memory address for the program, your program will generate a **run-time error** during execution.

- The compiler does not check null references for you!

Example:



Calling Object Methods

A method is called on an object by supplying an object reference and the name and parameters of the method.

Syntax:

objectReference.*methodName*(*parameters*)

Remember:

- Each object has its own class which defines which methods it can perform.
- Each object has its own area of memory storing its data.
- An object reference is a pointer to a particular object in memory, so Java knows which object we are talking about by providing the object reference.
  - ⇨ The object reference is called an ***implicit parameter***.

Creating and Using Objects
Calling Object Methods Example

```
public class TestBankAccount
{  public static void main(String []args)
   {  BankAccount savings = new BankAccount(100);
      BankAccount checking = new BankAccount(50);
      BankAccount myRef;   // No object allocated!

      System.out.println(savings.getBalance()); // 100
      System.out.println(checking.getBalance());// 50
      System.out.println(myRef.getBalance()); // Error!
      savings.deposit(50);
      checking.withdraw(40);
      myRef = savings;
      myRef.withdraw(20);
      System.out.println(savings.getBalance()); // 130
      System.out.println(checking.getBalance()); // 10
      System.out.println(myRef.getBalance()); // 130
      myRef = checking;
      myRef.deposit(50);
      System.out.println(myRef.getBalance()); // 60
   }
}
```

Advanced: Implicit Parameter this

When an object method is called, we tell Java which object to use based on an object reference.

This object reference is then accessible within an object method as the **this** reference.

Example:

```
public class TestThis
{  public static void main(String []args)
   {  BankAccount checking = new BankAccount(50);
      BankAccount savings = new BankAccount(100);

      System.out.println(savings.getBalance());
      // this reference set to savings
      System.out.println(checking.getBalance());
      // this reference set to checking
   }
}
```


Implicit Parameter this (2)

```
public class BankAccount
{  public void deposit(double amount)
   {  this.balance = this.balance + amount; }
   public void withdraw(double amount)
   {  this.balance = this.balance - amount; }
   public double getBalance()
   {  return this.balance; }

   // this points to the current object being used
   // Using this is optional because Java assumes you
   // are working with the current object

   public BankAccount()
   { this.balance = 0; }
   public BankAccount(double balance)
   { this.balance = balance; }

   private double balance;
}
```


Access Specifiers Public and Private

Question: A method in class X is defined as private. Can it access a public variable in class Y?

A) Yes

B) No

Access Specifiers Public and Private

Question: Which statement is true?

A) It is a good idea to make all instance variables public.

B) Every parameter should be declared as public or private.

C) A method in class X can call a private method in class Y.

D) A method in class X can access a private instance variable in class X.

Objects and Object References

Question: How many objects are created by this code?

```
BankAccount savings, checking;
BankAccount myAcct, myAcct2;

savings = new BankAccount();
myAcct = savings;
checking = new BankAccount();
```

A) 1
B) 2
C) 3
D) 4

Objects and Object References

Question: How much money is in the account referenced by the `myAcct2` object reference?

```
BankAccount savings, checking;
BankAccount myAcct, myAcct2;

savings = new BankAccount(50);
myAcct = savings;
checking = new BankAccount(100);
savings = checking;
myAcct2 = myAcct;
```

A) unknown
B) 50
C) 100
D) undefined

Practice Questions

1) Explain the difference between a class, an object, and an object reference.

2) Create a program which creates a new BankAccount object called savings with an initial balance of $100. Then, deposit $40, withdraw $20, and print the current balance.

3) Modify the BankAccount class to also store an interest rate.
- Allow the user to specify the interest rate in a constructor.
- Create a method for setting the interest rate.
- Create a method called calcInterest() to update the current balance based on the interest rate.
- Test your class with an account with $1000 and 10% interest rate. Deposit $100, calculate interest, and print balance.

Interfaces

Interfaces are used to allow a class to implement methods of another class without inheriting from it.

An ***interface*** is a class where:
- All methods are `public` and `abstract` (no implementation).
- All variables are `static` and `final`. (no instance variables).

A class which implements an interface must implement all methods of the interface.

A class can implement multiple interfaces.

Keyword to indicate implementing an interface is: **`implements`**

Interfaces Example

```
interface Shape {
   int numSides();
   int getArea();
}

class Square implements Shape {
   public int numSides() { return 4; }
   public int getArea()  { return len*height; }

   private int len;
   private int height;
}
```


Object - THE SUPERCLASS

The class **Object** in Java is the root of the inheritance hierarchy or the superclass of all classes.
- That is, every class defined in Java and that you define inherits from the `Object` class.
- If you define a class that does not inherit from another class, your class automatically extends the `Object` class.

The `Object` class has some defined methods:
- `String toString()`
  - ⇨ returns a string representation of the object
- `boolean equals(Object other)`
  - ⇨tests whether the object equals another object
- `Object clone()`
  - ⇨makes a full (or deep) copy of the object
    - does not just copy the object reference, copies the entire object

Overriding toString method

```
public class BankAccount
{
   public String toString()
   {  return "BankAccount[balance="+ balance+ "]"; }

   ...

   private double balance;
   private int accountNum;
   private static int lastAccountNum = 0; // Static
}
```

`toString()` method:
1) Returns a string representation of your object.

Casting and Method Access

It is possible to assign an object reference of a subclass to the an object reference variable of a superclass.
- This is allowed because a subclass is a special case of the superclass.
- However, you are unable to access any methods/variables in the subclass using a superclass object reference.
- All objects inherit from the **`Object`** class, so they can be assigned to an **`Object`** reference variable.

It is possible to explicitly cast an object reference variable for a superclass to a subclass variable only if the superclass variable references a valid subclass instance.
- Otherwise, a run-time error will result.

Subclass to Superclass Example

```
public class TestSubclass
{  public static void main(String[] args)
   {  BankAccount checking = new BankAccount(100);
      SavingsAccount savings = new SavingsAccount(10);
      BankAccount anyAccount;

      savings.deposit(50);
      anyAccount = checking;
      System.out.println(anyAccount.getBalance()); // 100

      // refer to subclass object using a superclass ref.
      anyAccount = savings;
      System.out.println(anyAccount.getBalance()); // 50

      savings.addInterest();     // legal call
      // addInterest call below does not work
      //  as it is not defined in BankAccount class
      anyAccount.addInterest();
   }
}
```


Object References Example

```
class Person{ private String name;
   public String getName() { return name; }
   public Person(String s) {name=s;} }

class Student extends Person { private String major;
   public String getMajor() { return major; }
   public Student(String n, String m)
   { super(n); major=m;} }

public class TestInheritance
{  public static void main(String[] args)
   {  Person p = new Person("Joe");
      Student s = new Student("Fred","Comp.Sci.");
      Object o=s;    // Yes-Object is superclass
      s=(Student) p; // No-Run-time error
      p=s;           // Yes-Person is superclass
      p.getName();   // Yes-available in Person class
      p.getMajor();  // No-getMajor()not in Person class
      s=(Student) p; // Yes - p refers to a Student obj.
   }
}
```


Variable Scope Overview

Depending on the type of variable, the period of existence of the variable, called its ***lifetime***, will change.

The ***lifetime*** of a variable is based on when the variable is created and how long it stays around in the program.

- When a variable is first defined in a program its lifetime begins.
- When a variable exits ***scope*** its lifetime ends.

The ***scope*** of a variable is the part of the program where you can access or use the variable.

Variable Scope Variable Types

There are four basic variable types in Java:

- 1) ***Instance variables*** - are variables that are defined in a class and are part of an object.
- 2) ***Static variables*** - are variables in a class which are common to all object instances. Only one copy of variable for all objects.
  - ⇨ Note that a static variable exists in a separate memory area and not within any particular object instance. Use keyword `static`.
- 3) ***Local variables*** - are variables defined in methods.
- 4) ***Parameter variables*** - are variables passed to methods to help them perform their computation.

Variable Scope Scope of Variable Types

The scope of variables depends directly on their type:

- 1) ***Instance variables*** - are created when an object instance is created using the `new` method. Instance variables are defined as long as there is at least one reference to the object in your program which is still in scope.
- 2) ***Static variables*** - are created when the class they are defined in is first loaded and are defined until the class is unloaded.
  - ⇨ This means static variables are around for the duration of your program.
- 3) ***Local variables*** - are created when the program enters the block in which they are defined and destroyed when the program exits that block.
  - ⇨ A variable defined in brackets ("{","}") is accessible anywhere within the block including nested blocks.
- 4) ***Parameter variables*** - are created when a method is first called and are destroyed when a method returns.

Variable Scope Variable Scope Rules

1) A variable defined in a block outlined using brackets is accessible within the block and any subblocks. Example:

```
public static void main(String[] args)
{  int i;
   {  int j;
      ...       // i & j accessible here
   }            // j goes out of scope
   ...          // only i accessible here
}
```

2) Two variables of the same name cannot be declared in the same scope.

```
public static void main(String[] args)
{  int i;
   ...
   double i;   // Not allowed i is already defined
}
```

Variable Scope
Scope Example

```
public class MethodScope
{  public static void main(String[] args)
   {  double amount = 25;  // amount defined in main()
      BankAccount acct = new BankAccount(100);
      acct.withdraw(amount); //amount in main() copied to
      // amt in withdraw() - Not same variable!
   }  // amount, acct go out of scope
      // Object acct can be deleted with variable balance
}

class BankAccount
{  public void withdraw(double amt)
   {  if (amt <= balance)
      {  double newBalance = balance - amt;
         // newBalance is only defined within brackets
         balance = newBalance;
      }  // newBalance goes out of scope and is deleted
   }  // method variable amt goes out of scope

   private double balance; // instance variable
}
```


Variable Scope
Common Scope Errors

1) Variables with the same name in different scopes are different variables!

```
public static double area(Rectangle rect)
{  double r = rect.getWidth() * rect.getHeight();
   return r;
}

public static void main(String[] args)
{  Rectangle r = new Rectangle(5, 10, 20, 30);
   double a = area(r);
}
```

The `double r` in method `area` is a different variable than the variable `Rectangle r` in `main` as they have different scopes.

Variable Scope
Common Scope Errors (2)

2) Beware of scope issues when declaring variables in for loops!

```
public class TestForScope
{  public static void main(String[] args)
   {  for (int i=1; i <= 5; i++)
      {  System.out.println(i);  // 1,2,3,4,5
      }
      // for loop has its own copy of variable i
      // i in for loop goes out of scope
      System.out.println(i);  //Not allowed- i is gone!
   }
}
```


Variable Scope
Advanced Topic: Shadowing

Shadowing occurs when a variable in an inner scope overrides or shadows a variable in an outer scope with the same name.

- Typically, shadowing is an unintended programming mistake.
- Shadowing is possible with variables of different types.

Example:

```
public class Coin
{  public Coin(double aValue, String aName)
   {  value = aValue;
      String name = aName; // Shadows name in class
   }
   ...
   private double value;
   private String name;       // name defined here
}
```

In the example, the programmer accidentally redeclared the string variable name in the constructor which overrides the instance variable in the class.

Advanced:
Method Parameters: Pass-by-value

All method parameters are passed to a method by **value** which means that even if they are changed in a method, they are not updated in the caller method.

To return a value from a method:

- 1) Return a single value using a return type.
- 2) Pass object references to the method which allow object values to be changed.

Note: Although you cannot change the value of any parameters, by passing object references which have access to objects, you can change object data.

- However, you cannot change the object reference value itself.

Variable Scope
Advanced Topic: Garbage Collection

Have you ever wondered what happens to objects that you no longer need after you created them using **new**?

- Unlike other languages, a Java programmer is not responsible for deleting or destroying objects that you no longer use.
- When an object has no valid references to it, Java may delete the object in memory in a process called ***garbage collection***.

The lifetime of an object in memory:

- 1) The object is created using `new` and a reference to its location in memory is created.
- 2) The object may have multiple object references during the program execution.
- 3) When all object reference variables go out of scope, the object has no more references and is marked for deletion.
- 4) Java periodically scans memory and deletes objects.

Variable Scope Practice Questions

With this code explain the lifetime and scope of all variables.

```
public class VariableScope
{  public static void main(String[] args)
   {  double amount = 25;
      BankAccount acct = new BankAccount(200);
      for (int i=1; i <= 3; i++)
         acct.deposit(amount);
      System.out.println(acct.getBalance());  // 125.0
   }

   private void doNothing(double a)
   {  int i = 5; return;   }

   public static final int MYNUM = 25;
}
```


Variable Scope Practice Questions (2)

```
class BankAccount
{  public void deposit(double amount)
   {  if (amount <= balance)
      {  double newBalance = balance - amount;
         balance = newBalance;
      }
      double balance = 50;
   }
   public double getBalance()
   {  return balance; }

   public BankAccount(double b)
   {  balance = b; lastAccountNum++;
      accountNum = lastAccountNum;
   }

   private double balance;
   private int accountNum;
   private static int lastAccountNum = 0;
}
```


Conclusion

Key object-oriented terminology:

- ***Object*** – an instance of a class.
- ***Class*** – an object template with methods and properties.
- ***Method*** – a set of statements that performs an action.
- ***Parameter*** – data passed into a method.
- ***Properties*** – are attributes of objects.

Access specifiers limit what methods can access.

Inheritance is a mechanism for creating a new class by extending the features of an existing class.

Object references point to objects in memory. Use `new` to create objects. Methods are called using an object reference.

The scope and lifetime of a variable depends on its type (instance, static, local, parameter).

Objectives

Definitions: class, object, method, parameter, instance variable, inheritance, superclass, subclass, interface

Java skills:

- Defining a class with variables and methods.
- Meaning of access specifiers: public, protected, private.
- Inheritance using `extends` and method/attribute inheritance.
- Creating objects using `new`.
- Purpose, use, and definition of constructors.
- **Difference between objects and object references.**
- Calling methods using object references. Implicit parameter this.
- Parameters are pass by value.
- Variable scope and lifetime for variable types.
- Advanced topics: shadowing, garbage collection