University of Dar es Salaam

CS 234: Object-Oriented Programming in Java

Lecture – Object Oriented Design & Concepts

Aron Kondoro

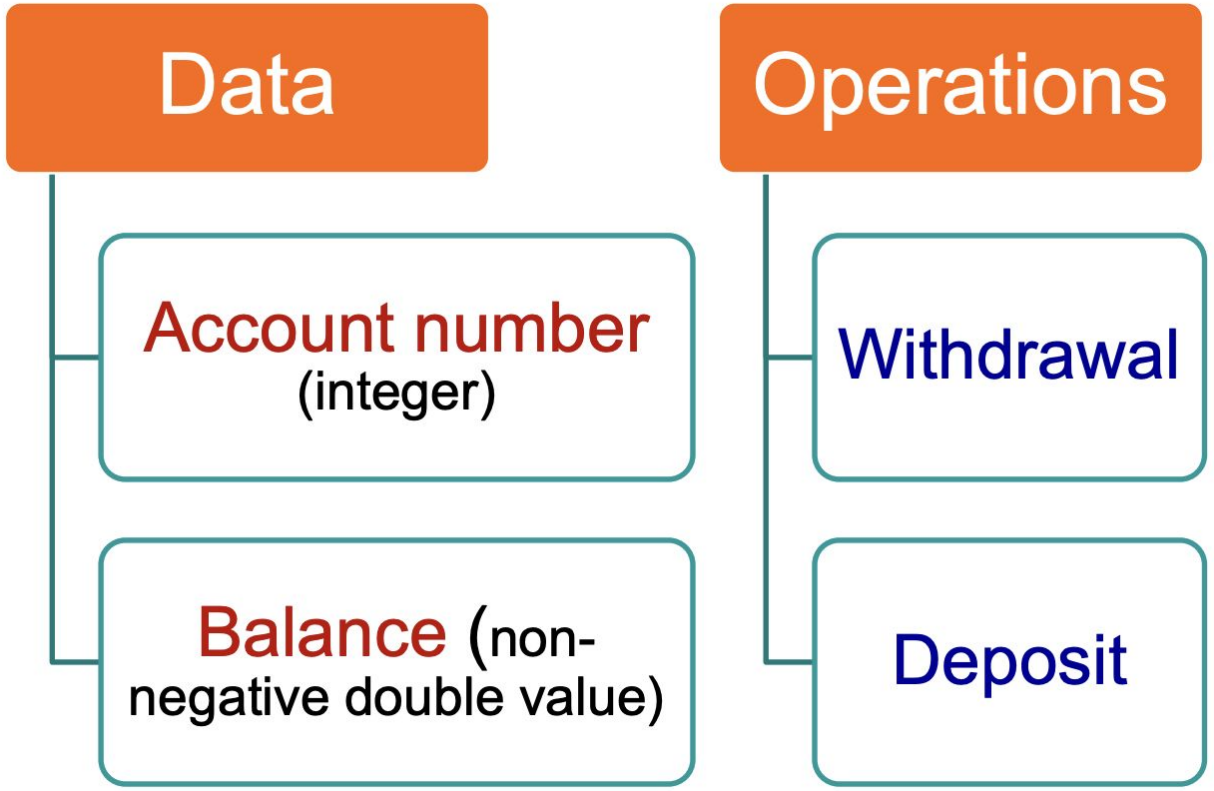
Outline

- **Object-Oriented Programming Model**
- **Encapsulation/Access Control**
- **Inheritance**
- Polymorphism
- Method Overriding
- Abstraction
- Interfaces

OO Programming Model

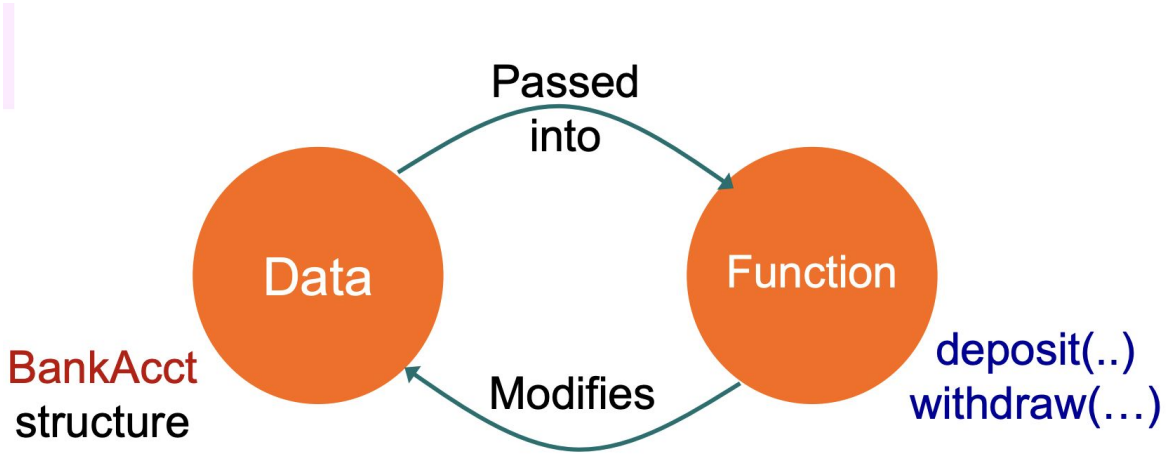
Example

- Design a program that models a bank account



Bank Account (Procedural Implementation)

- Data (structure) and functions (operations) are treated as separate entities
- Program is viewed as transforming data



Bank Account (Procedural Implementation)

```
typedef struct {
   int acctNum;
   double balance;
} BankAcct;
```

Structure to hold data

```
void initialize(BankAcct *baPtr, int anum) {
   baPtr->acctNum = anum;
   baPtr->balance = 0;
}

int withdraw(BankAcct *baPtr, double amount) {
   if (baPtr->balance < amount)
      return 0;  // indicate failure
   baPtr->balance -= amount;
   return 1;      // indicate success
}

void deposit(BankAcct *baPtr, double amount)
{  ... Code not shown ... }
```

Functions to provide basic operations

Correct use of BankAcct and its operations

```
BankAcct ba1;

initialize(&ba1, 12345);
deposit(&ba1, 1000.50);
withdraw(&ba1, 500.00);
withdraw(&ba1, 600.00);
      ...
```

Wrong and malicious exploits of BankAcct

```
BankAcct ba1;

deposit(&ba1, 1000.50);

initialize(&ba1, 12345);
ba1.acctNum = 54321;

ba1.balance = 10000000.00;
       ...
```

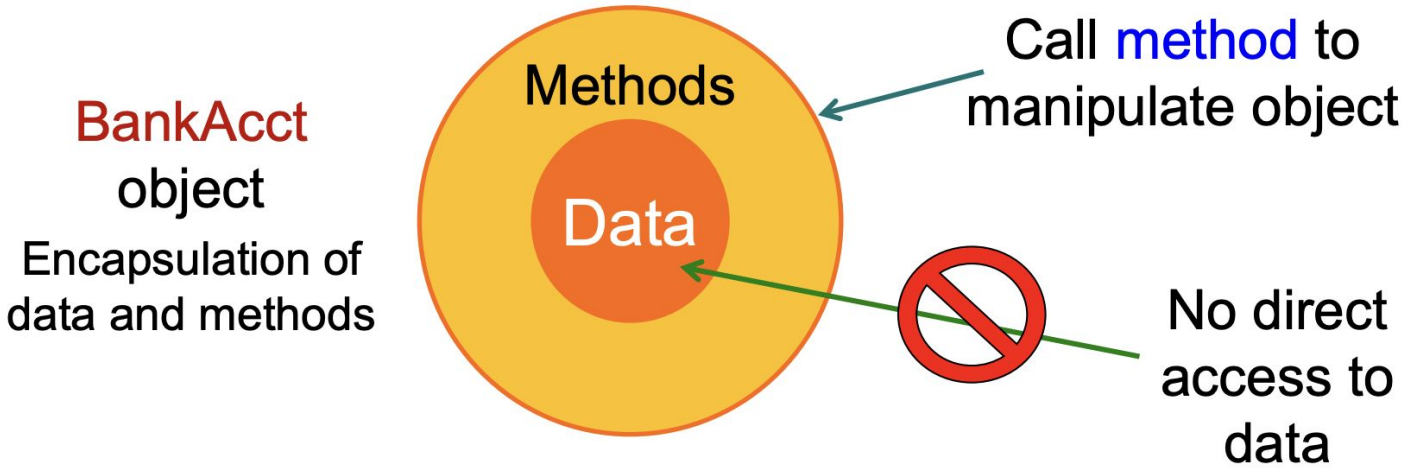
Forgot to initialize

Account Number should not change!

Balance should be changed by authorized operations only

Bank Account (OO Implementation)

- Program is viewed as a collection of **objects**
  - Computation is done via object interaction
- Each object has a set of **attributes** and **functionalities**
  - Functionalities are generally exposed
  - Attributes are generally hidden



Procedural (e.g. C)

- Procedural/Imperative
  - View a program as a process of transforming data
  - Data and associated functions are separated
  - Data is publicly accessible to everyone
- Advantages
  - Resembles execution model of computer
  - Less design overhead
- Problems
  - Harder to understand (unclear relation between data and functions)
  - Hard to maintain
  - Hard to extend

OOP (e.g. Java)

- Concepts: encapsulation, inheritance, abstraction, and polymorphism
- Advantages
  - Easier to design (resembles the real world)
  - Easier to maintain (enforces modularity)
  - Extensible
- Problems
  - Reduced execution efficiency
  - Longer code (higher design overhead)

Fundamental OOP Concepts

- **Encapsulation**
  - Bundling data and associated functionality
  - Hiding internal details and restricting access
- **Inheritance**
  - Deriving one class from another (code reuse)
- **Abstraction**
  - Hiding the complexity of the implementation
  - Focusing on specifications instead of implementation details
- **Polymorphism**
  - Changes in functionality depending on data type

University of Dar es Salaam

Encapsulation

Access Control

Encapsulation

- Wrapping up of data under a single unit
- It is the mechanism that binds together **code** and the **data** it manipulates
- It is a protective shield that prevents data from being accessed by outside code
  - Protect the data stored in a class from system-wide access

Encapsulation

- Implemented by making fields **private** and accessing only through member functions of own classes
- These special member functions for accessing private variables are called **getter** and **setter** methods
- **Getter methods** return the field
- **Setter methods** changes the value of the field

Example (without Access Control)

```
public class CreditCard {
   String cardNumber;
   double expenses;
   void charge(double amount) {
      expenses = expenses + amount;
   }
   String getCardNumber(String password) {
      if (password.equals("SECRET!3*!")) {
         return cardNumber;
      }
         return "jerkface";
   }
}
```

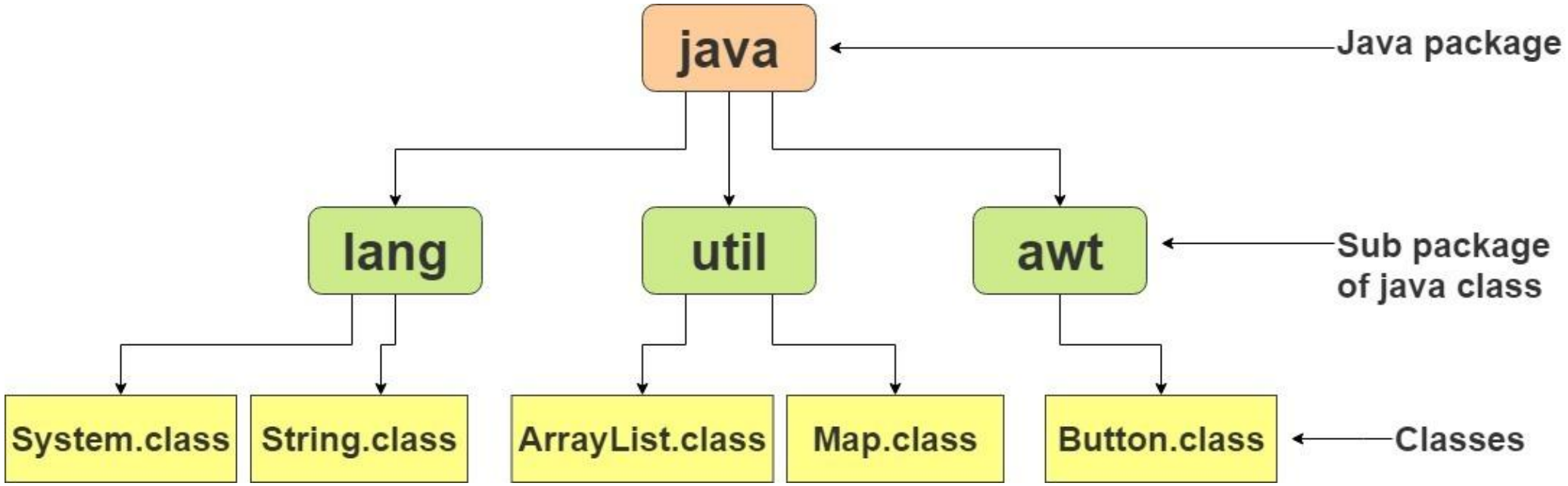
Malicious Class

```
public class Malicious {
   public static void main(String[] args) {
      maliciousMethod(new CreditCard());
   }
   static void maliciousMethod(CreditCard card)
   {
      card.expenses = 0;
      System.out.println(card.cardNumber);
   }
}
```

Java Packages

- A group of similar types of classes, interfaces & sub-packages
- Two types: **built-in** & **user-defined**
  - Built-in examples: java, lang, awt, javax, swing, net etc…
- Benefits
  - Categorize classes and interfaces for easy maintenance
  - Access protection
  - Prevents naming collisions

Built-in Java Packages



Java Package Example

- The **package** keyword is used to create a package in Java

```
package packA;

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

How to access class in a package

- import package.*;
- import package.classname;
- fully qualified name

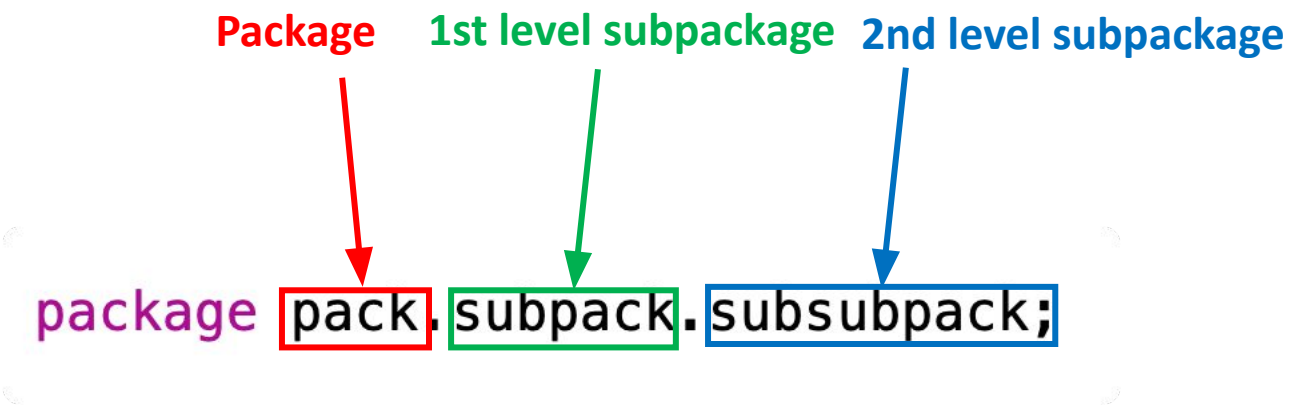
```
package packB;
import packA.*;

class B {
   public static void main(String args[]){
      A obj = new A();
      obj.msg();
   }
}
```

```
package packB;
import packA.A;

class B {
   public static void main(String args[]){
      A obj = new A();
      obj.msg();
   }
}
```

```
package packB;

class B {
   public static void main(String args[]){
      packA.A obj = new PackA.A();
      obj.msg();
   }
}
```

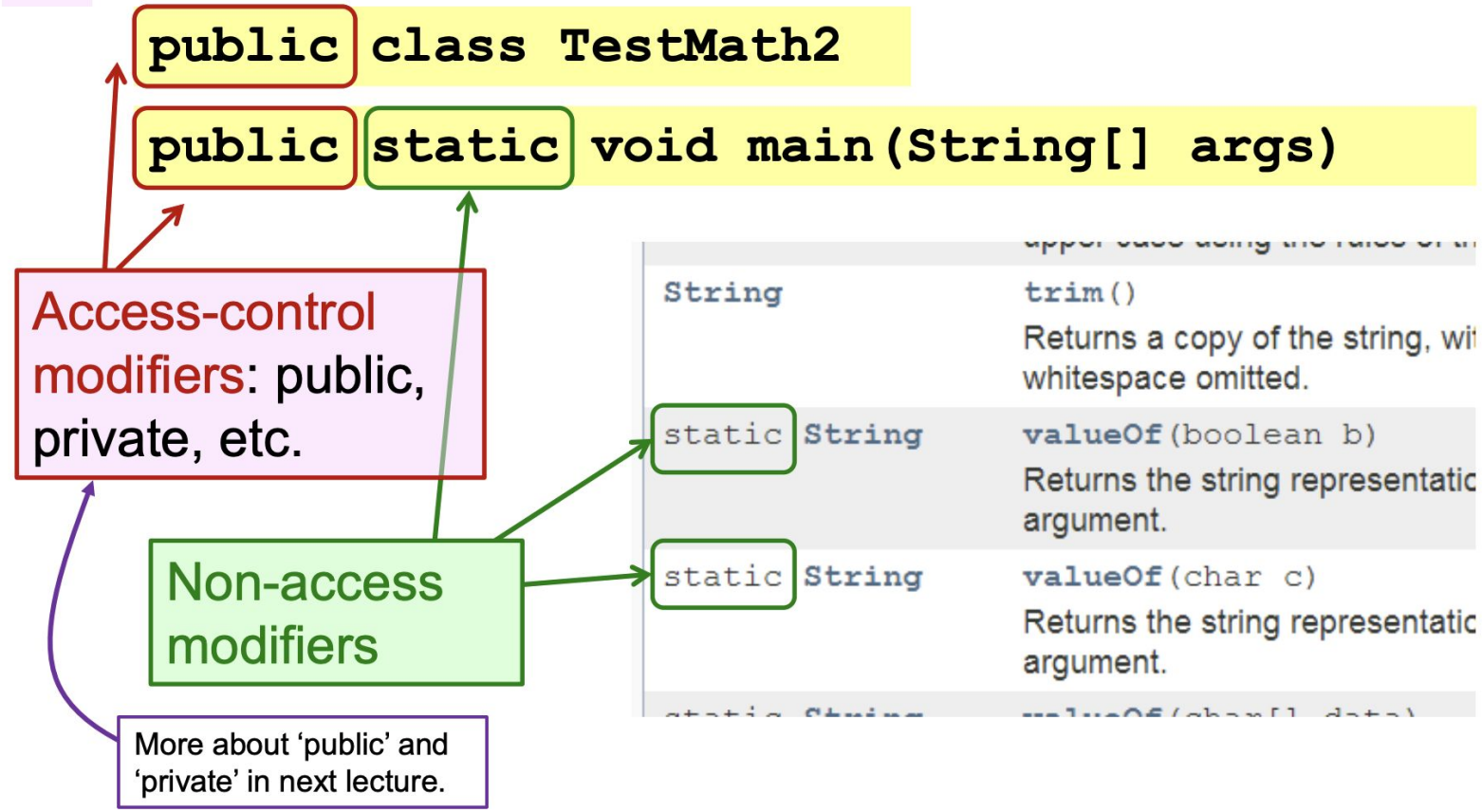
Java Sub-packages

- Packages inside other packages are called **sub-packages**



Modifiers

- Keywords added to specify how a class/attribute/method works



Access Modifiers

- Specify accessibility or scope of a field, method, constructor or class
- Types
  - **Private**: accessible only within the class
  - **Default (none)**: accessible only within the package
  - **Protected**: accessible within the package and outside via a child class (not accessible outside if no child class
  - **Public:** accessible everywhere (within/outside class or package)

Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Example (with Access Modifiers)

```
public class CreditCard {
   private String cardNumber;
   private double expenses;
   public void charge(double amount) {
      expenses = expenses + amount;
   }
   public String getCardNumber(String password)
   {
      if (password.equals("SECRET!3*!")) {
         return cardNumber;
      }
      return "jerkface";
   }
}
```

Why Access Modifiers?

- Protect private information
- Clarify how to use the class
- Keep implementation separate from interface

Access Modifier Guidelines

- **Attributes** are usually **private**
  - Information hiding – shield object data from outside view
  - Provide public methods for accessing attributes
  - Exceptions – **Point** class has public attributes ***x*** and ***y***
- **Methods** are usually **public**
  - So that they are available to users
  - Internal methods in service of the class should be declared **private**

How to Access?

- Using getter and setter methods

```
class Animal {
   private String name;
   private double averageWeight;
   private int numberOfLegs;

   // Getter methods
   public String getName() {
      return name;
   }
   public double getAverageWeight() {
      return averageWeight;
   }
   public int getNumberOfLegs() {
      return numberOfLegs;
   }

   // Setter methods
   public void setName(String name) {
      this.name = name;
   }
   public void setAverageWeight(double averageWeight) {
      this.averageWeight = averageWeight;
   }
   public void setNumberOfLegs(int numberOfLegs) {
      this.numberOfLegs = numberOfLegs;
   }
}
```

Inheritance

Introduction

- The process by which one class acquires the **properties** and **functionalities** of another class
  - **Super (Base) Class**: the class whose features are inherited
  - **Sub (Child) Class**: the class that inherits the other class
- Inheritance enables code reusability
  - Each subclass defines only those features that are unique to it. The rest of the features can be inherited from the parent class.
- Code in the base class need not be rewritten in the child class

Introduction

- Each subclass can be a superclass of future subclasses
- A subclass can add its own **fields** and **methods**
- A subclass is **more specific** that its superclass and represents a **more specialized** group of objects
- The subclass exhibits the behaviour of its superclass and can **add its own behaviours** specific to the subclass

Examples

- **Superclasses** tend to be *more general* and **subclasses** *more specific*

Superclass	Subclasses
Student	GraduateStudent, UndergraduateStudent
Shape	Circle, Triangle, Rectangle, Sphere, Cube
Loan	CarLoan, HomeImprovementLoan, MortgageLoan
Employee	Faculty, Staff
BankAccount	CheckingAccount, SavingsAccount

Syntax: Inheritance in Java

- To inherit the class we use the **extends** keyword
- Class **A** is the **child** class and class **B** is the **parent** class

```
class A extends B {

}
```

Creating a subclass

- Consider an existing **BankAccount** class

```
class BankAccount {

   private int accNumber;
   private double balance;

   public BankAccount() {}

   public BankAccount(int acc, double bal) {}

   public int getAccNumber() {}

   public double getBalance() {}

   public boolean withdraw(double amount) {}

   public void deposit(double amount) {}

   public void print() {}

}
```

Creating a subclass

- Let's define a new **SavingAccount** class
- Basic information
  - Account number, balance
  - **Interest rate (new requirement)**
- Basic functionality
  - Withdraw, deposit
  - **Pay interest (new requirement)**
- Compare **SavingAccount** vs **BankAccount**
  - SavingAccount shares more than 50% of the code with BankAccount

Creating a subclass

- Duplicating code is **undesirable** as it is hard to maintain
  - Need to correct all copies if errors are found
  - Need to update all copies if modifications are required
- Duplicating code logically separates them
  - Two classes that share this code are still logically unrelated
- We should create SavingAccount as a **subclass** of BankAccount

Creating a subclass

```
class BankAccount {

   protected int accNumber;
   protected double balance;

   ...

}
```

The **protected** keywords allows Subclass to access the attribute directly

```
class SavingAccount extends BankAccount {

   protected double rate; // interest rate

   public void payInterest() {
      balance += balance * rate;
   }

}
```

The **extends** keyword indicates inheritance

This allows the subclass of SavingAccount to access the rate. If this is not intended, you may change it to private.

Benefits of OO Inheritance

- Can **save time** during program development by basing new classes on existing proven and debugged high quality software
- Reduces the amount of redundant code
  - In SavingAccount class – no definition of `accNumber` & `balance`, no definition of `withdraw()` and `deposit()`
- Improves **maintainability**
  - E.g. if a method is modified in BankAccount class, no changes are needed in SavingAccount class
- Reduces unintended changes
  - The code in BankAccount class remains untouched
    - Other programs that depend on BankAccount are unaffected

Constructors in subclasses

- Unlike normal methods, constructors are **NOT** inherited
  - You need to define constructor(s) in subclasses

```
class SavingAccount extends BankAccount {

   protected double rate; // interest rate

   public SavingAccount(int acc, double bal, double rate) {
      accNumber = acc;
      balance = bal;
      this.rate = rate;
   }

   ...

}
```

Using the subclass

```
public class TestSavingAccount {

    public static void main(String[] args) {
        SavingAccount sa1 = new SavingAccount(2, 10000.0, 0.05);

        sa1.print();
        sa1.withdraw(50.0);

        sa1.payInterest();
        sa1.print();
    }

}
```

Method inherited from BankAccount

Method in SavingAccount

The “super” keyword

- The **super** keyword allows us to use the methods (including constructors) in the superclass directly

```
public class SavingAccount extends BankAccount {

    ...

    public void print() {
        super.print();
        System.out.printf("Interest: %.2f%%\n", getRate());
    }

}
```

The "super" keyword

- The **super** keyword allows us to use the methods (including constructors) in the superclass directly
- If you use the superclass' constructor, it must be the **first statement** in the method body

```
class SavingAccount extends BankAccount {

  protected double rate; // interest rate

  public SavingAccount(int acc, double bal, double rate) {
    super(acc, bal);
    this.rate = rate;
  }

  ...

}
```

Method Overriding

- Sometimes we need to modify the inherited method:
  - To change/extend the functionality
  - this is called **method overriding**
- In the `SavingAccount` class:
  - The `print()` method inherited from `BankAccount` should be modified to include the interest rate in the output
- To override an inherited method:
  - Re-implement the method in the subclass using the same method header
  - **Method header** refers to the name and parameters type of the method (also known as **method signature**)
  - The overridden method cannot be more restrictive

Example: Method Overriding

```
public class SavingAccount extends BankAccount {

    ...

    public void print() {
        System.out.println("Account Number: " + getAccNumber());
        System.out.printf("Balance: Tsh%.2f\n", getBalance());
        System.out.printf("Interest: %.2f%%\n", getRate());
    }

}
```

Subclass as substitute

- An added advantage of inheritance is that:
  - Whenever a superclass object is expected, a subclass object is **acceptable as a substitution**
    - Caution: the **reverse is NOT true** (E.g. A cat is an animal but an animal may not be a cat.)
  - Hence, all existing functions that work with the superclass objects will work on subclass objects with no modification

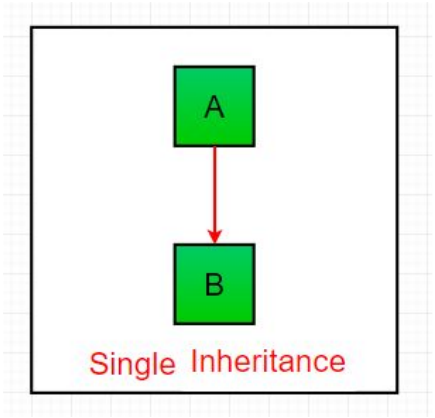
Example: subclass as substitute

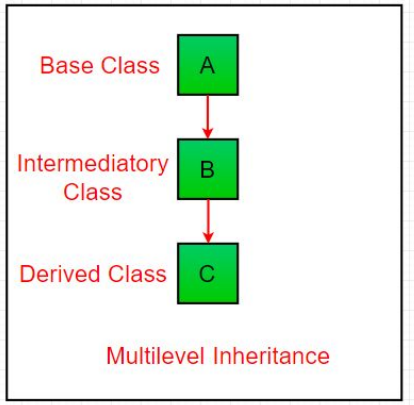
```
public class TestAccountSubclass {

    public static void transfer(BankAccount fromAcc, BankAccount toAcc, double amount) {
        fromAcc.withdraw(amount);
        toAcc.deposit(amount);
    }

    public static void main(String[] args) {
        BankAccount ba = new BankAccount(1, 2500.00);
        SavingAccount sa = new SavingAccount(2, 5000.0, 0.05);

        transfer(ba, sa, 500.00);

        ba.print();
        sa.print();
    }

}
```

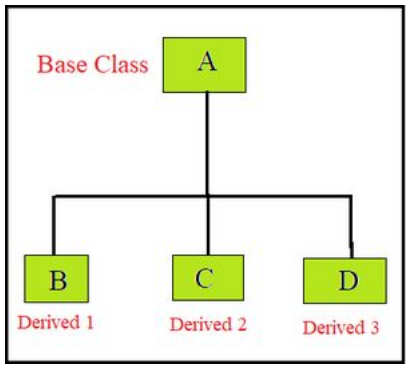
`transfer()` method also works on the `SavingAccount` object sa

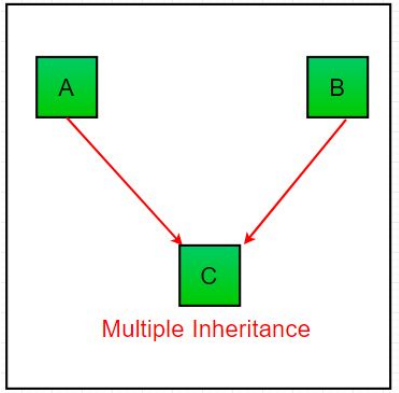
Types of Inheritance

- **Single Inheritance**: child and parent class relationship where a class extends the another class.
- **Multilevel inheritance**: child and parent class relationship where a class extends the child class. For example class A extends class B and class B extends class C.
- **Hierarchical inheritance**: child and parent class relationship where more than one classes extends the same class. For example, class B extends class A and class C extends class A.
- **Multiple Inheritance**: one class extending more than one classes, which means a child class has two parent classes.









Direct vs Indirect Superclass

- The **direct superclass** is the superclass from which the subclass explicitly inherits
- An **indirect superclass** is any class above the direct superclass in the class hierarchy
- The Java class hierarchy begins with the class `Object` (in java.lang)
  - Every class in Java directly or indirectly **extends** Object
  - Any methods that work with the Object reference will work on **objects of any class**
  - Two inherited `Object` methods are: `toString()` and `equals()`

Inheritance in Java

- In Java, **only single inheritance is allowed**
  - Multiple inheritance: C++, Python, Perl, Scale etc..
- Java's alternative to multiple inheritance can be achieved through the use of interfaces - to be covered later. A Java class may implement multiple interfaces.)

is-a vs has-a relationship

- **Caution**: Do not overuse inheritance
- The subclass-superclass relationship is known as an *is-a* relationship
  - Use the *is-a* as the rule of thumb
  - `SavingAccount` **is-a** `BankAccount`
- Frequently confused with the *has-a* relationship
- *Is-a* represents **inheritance**
  - In an is-a relationship, an object of a subclass can also be treated as an object of its superclass
- *Has-a* represents **composition**
  - In a *has-a* relationship, an object contains as members references to other objects

Preventing Inheritance

- Sometimes, we want to prevent inheritance by another class
  - e.g. To prevent a subclass from corrupting the behaviour of its superclass
- Use the **final** keyword
  - E.g. `final class SavingAccount` will prevent a subclass to be created from `SavingAccout`
- Sometimes, we want a class to be inheritable, but want to prevent some of its methods to be overridden by its subclass
  - Use the **final** keyword on the particular method:
    - `public final void payInterest() { ... }`
  - This will prevent the subclass of `SavingAccount` from overriding `payInterest()`

University of Dar es Salaam

Quiz 1

```
public class ClassA {

    protected int value;

    public ClassA() {

    }

    public ClassA(int val) {
        value = val;
    }

    public void print() {
        System.out.println("Class A: value = " + value);
    }

}
```

```
public class ClassB extends ClassA {

    protected int value;

    public ClassB() {}

    public ClassB(int val) {
        super.value = val - 1;
        value = val;
    }

    public void print() {
        super.print();
        System.out.println("Class B: value = " + value);
    }

}
```

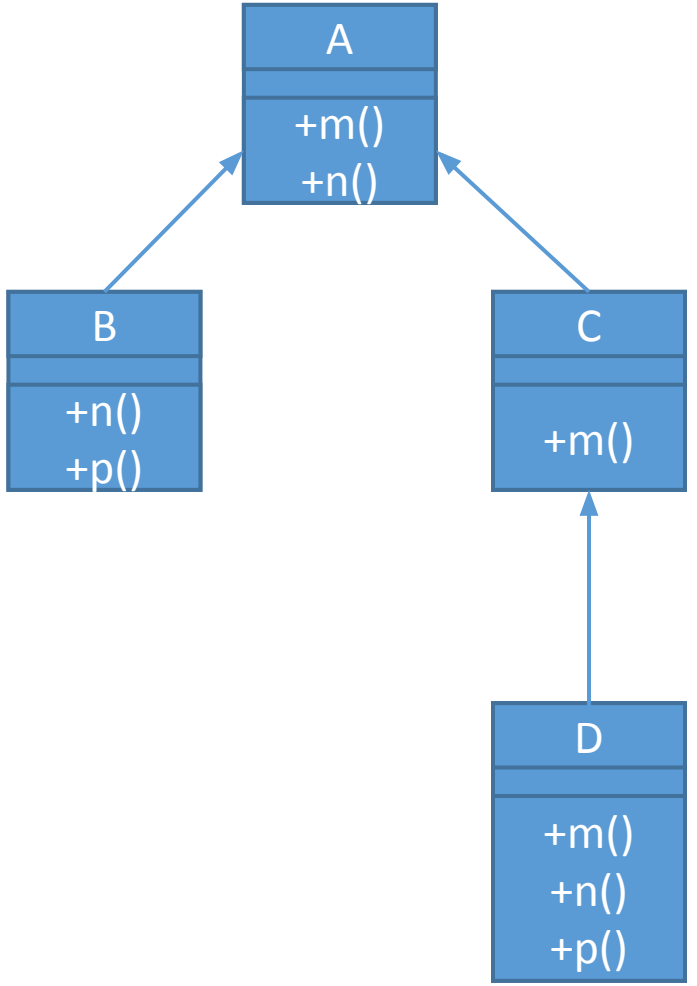
```
public class ClassC extends ClassB {

    private int value;

    public ClassC() {}

    public ClassC(int val) {
        super.value = val - 1;
        value = val;
    }

    public void print() {
        super.print();
        System.out.println("Class C: value = " + value);
    }

}
```

```
Class A: value = 123
----------
Class A: value = 455
Class B: value = 456
----------
Class A: value = 0
Class B: value = 788
Class C: value = 789
```

```
public class TestSubClasses {

    public static void main(String[] args) {
        ClassA objA = new ClassA(123);
        ClassB objB = new ClassB(456);
        ClassC objC = new ClassC(789);

        objA.print();
        System.out.println("----------");
        objB.print();
        System.out.println("----------");
        objC.print();
    }

}
```

Quiz 2

- Assume all methods print out a message of the form <class name>.<method name>
  - E.g. method m() in class A prints put “A.m”
- If a class overrides an inherited method, the method’s name will appear in the class icon. Otherwise, the inherited method remains unchanged in the subclass
- For each code fragment in the table that follows, indicate whether
  - The code will cause a compilation error, and briefly explain why?
  - The code can compile and run. Supply the execution result



Quiz 2

Code Fragment	Compilation Error? Why?	Execution Result
A a = new A(); a.m();		A.m
A a = new A(); a.k()	Method k() is not defined in class A	
A a = new C(); a.m();		
B b = new A(); b.n();		
A a = new B(); a.m();		
A a; C c = new D(); A = c; a.n()		
B b = new D(); b.p();		
C c = new C(); c.n();		
A a = new D(); a.p();		

Outline

- Object-Oriented Programming Model
- Encapsulation/Access Control
- Inheritance
- **Polymorphism**
- **Method Overriding**
- **Abstraction**
- **Interfaces**

Polymorphism

Introduction

- Polymorphism
  - An important concept in object-oriented programming
  - Means **more than form**
  - The same entity (**method, operator or object**) can perform different operations in different scenarios

Example

- Superclass: `Polygon`
- Subclasses: `Square` and `Circle`
- The `render()` method is used to render the shape
  - However, the process differs between the square and circle
  - It behaves differently in different classes
- Hence, `render()` is **polymorphic**

```
class Polygon {

  // method to render a shape
  public void render() {
    System.out.println("Rendering Polygon...");
  }
}

class Square extends Polygon {

  // renders Square
  public void render() {
    System.out.println("Rendering Square...");
  }
}

class Circle extends Polygon {

  // renders circle
  public void render() {
    System.out.println("Rendering Circle...");
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Square
    Square s1 = new Square();
    s1.render();

    // create an object of Circle
    Circle c1 = new Circle();
    c1.render();
  }
}
```

Ways to achieve polymorphism in Java

- Method Overriding
- Method Overloading
- Operator Overloading

Method Overriding

- In the **parent-child** relationship, the method in the subclass **overrides** the same method in the superclass
- In this case, the same method will perform one operation in the superclass and another in the subclass
- The method that is called is determined during program execution
  - Hence, method overriding is called **run-time polymorphism**

```
class Language {
  public void displayInfo() {
    System.out.println("Common English Language");
  }
}

class Java extends Language {
  @Override
  public void displayInfo() {
    System.out.println("Java Programming Language");
  }
}

class Main {
  public static void main(String[] args) {

    // create an object of Java class
    Java j1 = new Java();
    j1.displayInfo();

    // create an object of Language class
    Language l1 = new Language();
    l1.displayInfo();
  }
}
```

Method Overloading

- In a Java class, we can create methods with the same name if they have different parameters
- This is known as **method overloading**
- The same method will perform different operations based on the parameter

```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

Polymorphism using method overloading

- The `display()` method performs different operations based on arguments passed
  - Prints pattern of * without arguments
  - Prints pattern of the parameter if a single `char` is passed
- The method that is called is determined by the compiler
  - Hence, method overloading is called **compile-time polymorphism**

```
class Pattern {

  // method without parameter
  public void display() {
    for (int i = 0; i < 10; i++) {
      System.out.print("*");
    }
  }

  // method with single parameter
  public void display(char symbol) {
    for (int i = 0; i < 10; i++) {
      System.out.print(symbol);
    }
  }
}

class Main {
  public static void main(String[] args) {
    Pattern d1 = new Pattern();

    // call method without any argument
    d1.display();
    System.out.println("\n");

    // call method with a single argument
    d1.display('#');
  }
}
```

Operator Overloading

- Some Java operators behave differently with different operands
  - For example, the + operator is overloaded to perform *numeric addition* and *string concatenation*
  - *&*, *|*, and *!* are overloaded for logical and bitwise operations

```
int a = 5;
int b = 6;

// + with numbers
int sum = a + b;  // Output = 11
```

```
String first = "Java ";
String second = "Programming";

// + with strings
name = first + second;  // Output = Java Programming
```

Polymorphic Variables

- A variable is called polymorphic if it refers to different values under different conditions
- Polymorphic variables are represented using object variables in Java
  - An object variable can refer to objects of its class or subclasses

```
class ProgrammingLanguage {
  public void display() {
    System.out.println("I am Programming Language.");
  }
}

class Java extends ProgrammingLanguage {
  @Override
  public void display() {
    System.out.println("I am Object-Oriented Programming Language.");
  }
}

class Main {
  public static void main(String[] args) {

    // declare an object variable
    ProgrammingLanguage pl;

    // create object of ProgrammingLanguage
    pl = new ProgrammingLanguage();
    pl.display();

    // create object of Java class
    pl = new Java();
    pl.display();
  }
}
```

Why Polymorphism?

- Allows us to create **consistent code**
  - Instead of writing different methods e.g. renderSquare() and renderCircle()
  - Enables us to program **in the general** instead of **in the specific**
  - The `print()` method is an example of polymorphism. It can print `char`, `int`, `string` etc
- A single variable can be used to store multiple data values
  - Value of inherited variable can be changed without affecting the value in the superclass
- We can design and implement **extensible** systems
  - New classes can be added with little or no modification
  - Only parts that require direct knowledge need to be altered

Abstraction

Introduction

- Abstraction is the feature of OOP that **shows** only essential information and **hides** unnecessary information
  - Hides unnecessary details from users
- Through abstraction, a developer ensures that only functionality is provided to the user
  - Implementation and other aspects are kept hidden to reduce complexity

Abstract Class

- An abstract class is a class that deals with the abstraction of our program
  - Declared using the **abstract** keyword
  - Cannot be instantiated
  - Is permitted to have both abstract and non-abstract methods
  - A class needs to be declared as an abstract class if it contains abstract methods
  - To use an abstract class, one needs to extend its child class and provide an implementation of all the abstract methods

Declaring an Abstract Class in Java

```
public abstract class Person {

}
```

- We cannot create an instance of this class

```
Person personInstance = new Person();  //not valid
```

Abstract Methods

- Abstract methods are methods without bodies or implementation
  - Meant to be used by abstract classes only
- We declare a method abstract by adding the abstract keyword in the method declaration. The method also ends with a semicolon.

```
public abstract class Person {

   public abstract void myJob();

}
```

```
public class Teacher extends Person {

   public abstract void myJob(){
      System.out.println("My job is Teaching.");
   }

}
```

Why Abstract Class?

- Main purpose of abstract classes is to function as base classes to be extended by their subclasses
- If a method will be overridden in all child classes there is no point in implementing it in the parent class
- By making this method abstract, we make it compulsory for all subclasses to implement this method
  - Otherwise, we will encounter a compilation error

Interfaces

What is an Interface?

- An interface is a **specification** of
  - a **required behaviour** – any class that **implements** the interface must perform the behaviour
  - **constant** data values
- An interface is a fully abstract class
  - It includes a group of abstract methods (methods without bodies)

Why use interfaces?

- Separate specification of behaviour from its implementation
  - Many classes can **implement** the same behaviour
  - Each class can implement the behaviour in its own way
- Benefits
  - The invoker of the behaviour **is not coupled** to the class that implements the behaviour (only knows the interface)
  - Enables polymorphism and code re-use

Writing an Interface

- An interface (before Java 8) may contain only:
  - abstract public methods
  - public static final variables (constants)
- You can omit the defaults
  - All methods are **public abstract**
  - All variables are **public static final**

```
public interface Valuable {
   public static final String CURRENCY = "Tsh";
   public abstract double getValue();
}
```

No method body

```
public interface Valuable {
   String CURRENCY = "Tsh";
   double getValue();
}
```

Automatically **public abstract**

Implementing an Interface

- A class that implements an interface must implement all the methods in the interface
- It may use any static constants in the interface

```
public static Money implements Valuable {
   private double value;

   public double getValue() {
      return value;
   }

   public String toString() {
      return Double.toString(value) + " " + CURRENCY;
   }
}
```

Capabilities of an Interface

- Interfaces can contain
  - public static final values (constants)
  - public instance method signatures (with no code)
  - Default implementations of methods (Java 8+)
- Interfaces **cannot** contain
  - static methods (prior to Java 8)
  - non-final or non-static variables
  - constructors

Limits on Interface use

- What you can do
  - Declare a class that implements an interface
  - Implement more than one interface
  - Define variables using interface types (or create an array or ArrayList using an interface as type)

```
// Iterator is an interface with type parameter
Iterator it = wordlist.iterator();
Valuable m = new Money(5, "Baht");
List mlist = new ArrayList();
```

Limits on Interface use

- You cannot
  - Create objects of an interface type
  - Access static behaviour using an interface type

```
Comparable cmp = new Comparable( ); // ERROR
Valuable m = new Valuable(); // ERROR
```

Example: Interface

- Many applications need to sort an array of objects
  - Requirement: one method that can sort almost anything
  - Question: what does the method need to know about the objects
  - Answer: needs a way to compare two objects
- The **Comparable** Interface
  - `compareTo()` return the result of the comparison
  - `a.compareTo(b) < 0` a should come before b
  - `a.compareTo(b) > 0` a should come after b
  - `a.compareTo(b) = 0` a and b have same order in the sequence

```
package java.lang;

interface Comparable {
   int compareTo(Object obj); //the required behaviour
}
```

Sorting using `Arrays.sort()`

- **Arrays** is a class in the package `java.util`
- It has a static `sort` method for arrays
- Uses the `Comparable` interface
- Sorts any kind of object that implements `Comparable`
- `Arrays.sort(Comparable[] a)`

```
String[] fruits = {"orange", "grapes", "apple", "mango", "banana"};
Arrays.sort(fruits);
for (String s : fruits)
   System.out.println(s);
```

`Comparable` Interface

- `Arrays.sort()` **depends only** on the `compareTo()` behaviour, not on any particular class
  - It can sort anything
- Enables code re-use
  - Any application can use `Arrays.sort` to sort an array
  - Any application can use `Collections.sort` to sort a list

Implement the `Comparable` Interface

```
public class Student implements Comparable {
    private String firstName;
    private String lastName;
    private long id;

    @Override
    public int compareTo(Object o) {
        Student other = (Student) o;
        if(other == null) return -1;
        if(this.id < other.id) return -1;
        if(this.id > other.id) return +1;
        else return 0;
    }
}
```

Problem with `Comparable`

- A problem with `Comparable` is that it accepts any Object
  - `student.compareTo(teacher)`
  - You have to do a lot of type checking

```
public int compareTo(Object other) {
   if (!(other instanceof Student))
      throw new IllegalArgumentException(". . ." );
   Student st = (Student) other;
   // now compare using Student st
}
```

Solution: Parameterized Interfaces

- In Java 5.0+ interfaces and classes can have **type parameters**
- a **Type parameter** is a variable that represents a type i.e. the name of a class or interface
  - E.g. the comparable interface has a type parameters (T) that is a placeholder for an actual data type
- The compiler will help to check compatibility i.e. this is called **type safety**

```
interface Comparable<T> {

   int compareTo(T obj);

}
```

Example: using Parameterized Interface

```
public class Student implements Comparable<Student> {
    private String firstName;
    private String lastName;
    private long id;

    @Override
    public int compareTo(Student other) {
        // No cast ot type-check needed
        if(other == null) return -1;
        return Long.compare(this.id, other.id);
    }
}
```

Java 8 Interfaces

- Java 8 interfaces **can contain code**
  - default methods (instance methods)
  - static methods

```
public interface Valuable {
   // abstract method that must be implemented
   public double getValue();

   // default method that is supplied by the interface
   default boolean isWorthless() {
      return this.getValue() == 0.0;
   }
}
```

Multiple Interfaces

- A class can also implement multiple interfaces
  - By separating the interface names by commas

```
public class MyApplication implements Comparable, Cloneable {

   // implement required behavior by Comparable
   public int compareTo(Object other) { ... }

   // implement behavior for Cloneable
   public Object clone( ) { ... }
   ...
}
```

Summary of Java Interfaces

- An interface can contain only:
  - static constants
  - instance method signatures (but no code).
- Implicit properties:
  - all methods are automatically public.
  - all attributes are automatically public static final.
- An interface may **NOT** contain
  - static (class) methods (**allowed in Java 8**)
  - implementation of methods (**allowed in Java 8**)
  - instance variables
  - constructor

Interfaces vs Abstract Classes

- **Type of variables:** Abstract class can have final, non-final, static and non-static variables. The interface has only static and final variables.
- **Implementation:** Abstract class can provide the implementation of the interface. Interface can't provide the implementation of an abstract class.
- **Inheritance vs Abstraction:** A Java interface can be implemented using the keyword "implements" and an abstract class can be extended using the keyword "extends".
- **Multiple implementations:** An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.
- **Accessibility of Data Members:** Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

Quiz 1

- The following Java application contains errors. Point out the statement(s) that contain errors. Explain what each of the errors is, and how it can be fixed.

```
public class OOPExercises {
   public static void main(String[] args) {
      A objA = new A();
      System.out.println("in main(): ");
      System.out.println("objA.a = "+objA.a);
      objA.a = 222;
   }
}
```

```
public class A {
    private int a = 100;
    public void setA( int value) {
       a = value;
    }
    public int getA() {
       return a;
    }
} //class A
```

Point out the error(s) and how they can be fixed.

Quiz 2

- Show the ouput of the following application

```
public class OOPExercises {
   public static void main(String[] args) {
      A objA = new A();
      B objB = new B();
      System.out.println("in main(): ");
      System.out.println("objA.a = "+objA.getA());
      System.out.println("objB.b = "+objB.getB());
      objA.setA (222);
      objB.setB (333.33);
      System.out.println("objA.a = "+objA.getA());
      System.out.println("objB.b = "+objB.getB());
   }
}
```

Output:

```
public class A {
    int a = 100;
    public A() {
       System.out.println("in the constructor of class A: ");
       System.out.println("a = "+a);
       a = 333;
       System.out.println("a = "+a);
    }
    public void setA( int value) {
       a = value;
    }
    public int getA() {
       return a;
    }
 } //class A
```

```
 public class B {
    double b = 123.45;
    public B() {
       System.out.println("-----in the constructor of class B: ");
       System.out.println("b = "+b);
       b = 3.14159;
       System.out.println("b = "+b);
    }
    public void setB( double value) {
       b = value;
    }
    public double getB() {
       return b;
    }
 } //class B
```