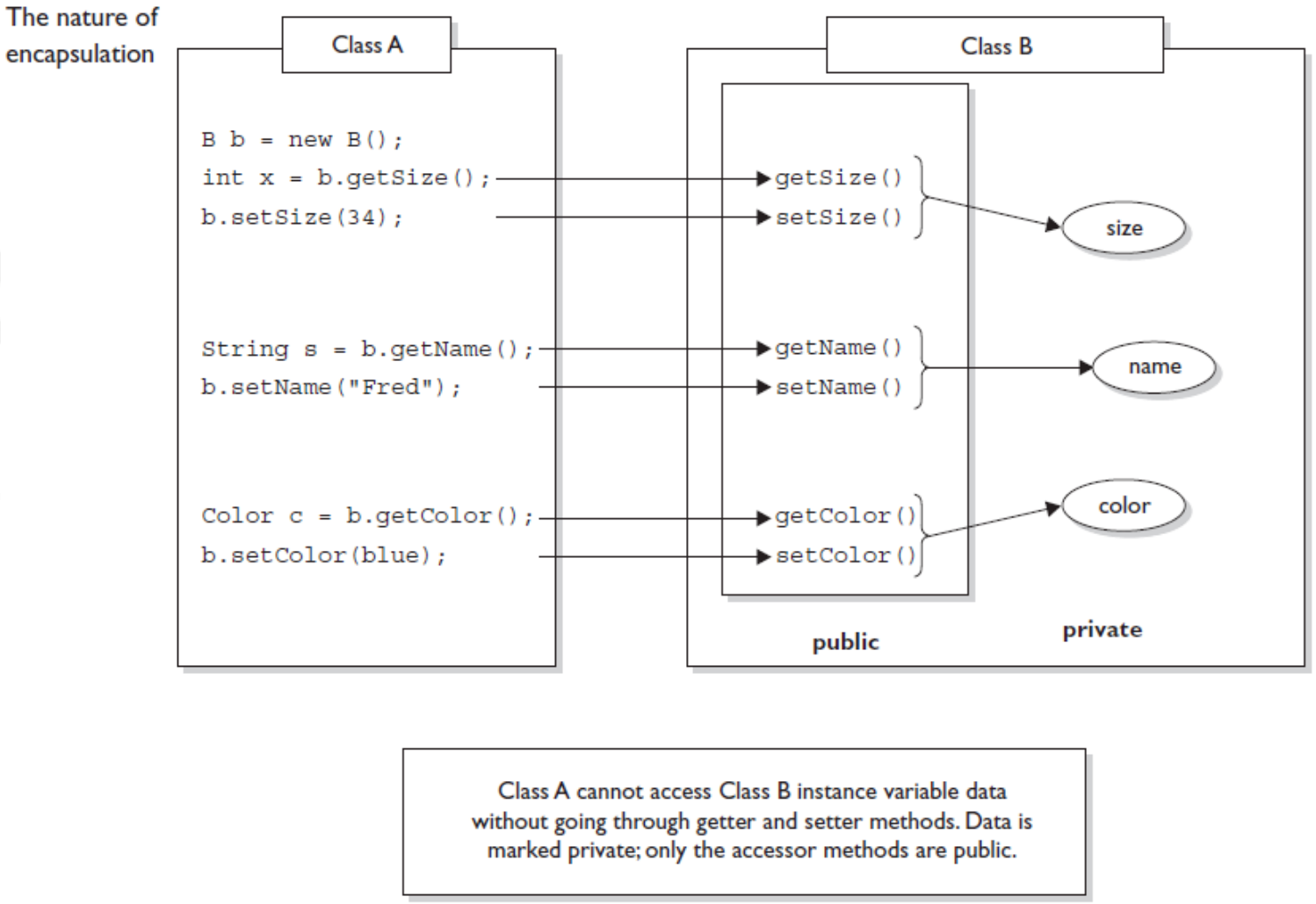
Object Orientation

Nilesh Dungarwal

Encapsulation

- Provides
  - flexibility
  - maintainability
- The ability to make changes in your implementation code without breaking the code of others who use your code is a key benefit of encapsulation

Nature of Encapsultion



Class A cannot access Class B instance variable data without going through getter and setter methods. Data is marked private; only the accessor methods are public.

Exam Watch

exam Watch

Look out for code that appears to be asking about the behavior of a method, when the problem is actually a lack of encapsulation. Look at the following example, and see if you can figure out what's going on:

```
class Foo {
   public int left = 9;
   public int right = 3;
   public void setLeft(int leftNum) {
      left = leftNum;
      right = leftNum/3;
   }
   // lots of complex test code here
}
```

Now consider this question: Is the value of right always going to be one-third the value of left? It looks like it will, until you realize that users of the `Foo` ***class don't need to use the*** `setLeft()` ***method! They can simply go straight to the instance variables and change them to any arbitrary*** `int` ***value.***

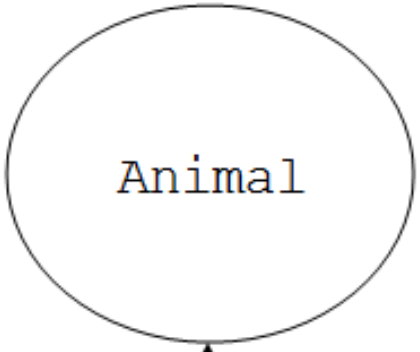
Inheritance

- To promote code reuse
- To use polymorphism

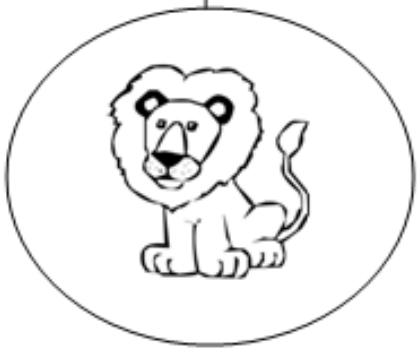
Review - inheritance

- Inheritance is a mechanism for defining a new class in terms of an existing class
- Inheritance allows the refinement or specialisation of an existing class
  - e.g. `Circle` is a subclass of `Shape`
  - inherited members can be redefined
  - Inherited members can not be hidden
  - new members can be added
- Inheritance is the key feature of object-oriented systems to promote software reuse
  - is the source “is a ” or “kind of” relationship

Subclassing



Kind Of



Attributes
- size
- colour

methods
- move
- eat
- reproduce

Animal

"extends"

Lion

Attributes
- size
- colour
- manecolour

methods
- move
- eat
- reproduce
- roar
- hunt

Inheritance in Java

- Define a new class as an extension of an existing class
  - potentially adds new data and methods
  - may redefine existing methods (polymorphism)

```
public class Animal {
private Color colour;
private int size;
public Animal(Color c, int s){
      colour=c;
      size=s;
}
...
public void eat(){
      //yum yum
}
...
}
```

```
public class Lion extends Animal{
      private Color maneColour;
      ...
      @Override
      public void eat(){
            //yum yum grow1...
      }
}
```

Method Inheritance

- A subclass inherits members from the super class
  - all non-private methods and variables of the parent are inherited
- Locally defined method hides method in parent
  - both methods are nevertheless available for the class
  - use the keyword `super` to explicitly refer to the super-class method

```
public class Lion extends Animal{

      @Override
      public void eat(){
            System.out.println("Lion eat method");
            super.eat();   //access overriden method eat
      }
}
```

Method Overriding and Polymorphism

- Overriding occurs when the subclass defines the same method as the super-class
  - method signatures must be identical
- Net result: Polymorphism!
  - class of object is determined at runtime, and the method in that class is called, i.e. dynamic binding
  - frequently used when working with collection of objects

Polymorphism

- Any Java object that can pass more than one IS-A test can be considered polymorphic
- Other than Object class all classes are polymorphic

Reference Variable

- A reference variable can be of only one type, and once declared, that type can never be changed (although the object it references can change).
- A reference is a variable, so it can be reassigned to other objects, (unless the reference is declared `final`).
- A reference variable's type determines the methods that can be invoked on the object the variable is referencing.
- A reference variable can refer to any object of the same type as the declared reference, or—this is the big one—**it can refer to any *subtype* of the declared type!**
- A reference variable can be declared as a class type or an interface type. If the variable is declared as an interface type, it can reference any object of any class that *implements* the interface.

Example

```
public class Animal {
	private Color colour;
	private int size;

	public void move(int distance, int direction){
		System.out.println("Generic move method");
	}
	public void eat(){
	System.out.println("Generic eat method");
	size++;
	}
}
```

```
Generic move method
Lion eat method
Generic eat method
```

```
public class Lion extends Animal{
	private Color maneColour;

	@Override
	public void eat(){
		System.out.println("Lion eat method");
		super.eat();//access overriden method eat
	}
	public void roar(){
		System.out.println("ROAAAAAAARRRRRRRR");
	}
	public void hunt(){
		System.out.println("Lion hunting method");
	}
}
```

```
Animal king = new Lion();
king.move(10,90);
king.eat();
```

Illegal Overrides

Illegal Override Code	Problem with the Code
`private void eat() { }`	Access modifier is more restrictive
`public void eat() throws IOException { }`	Declares a checked exception not defined by superclass version
`public void eat(String food) { }`	A legal overload, not an override, because the argument list changed
`public String eat() { }`	Not an override because of the return type, not an overload either because there's no change in the argument list

Overloaded Methods

- Overloaded methods let you reuse the same method name in a class, but with different arguments

- Overloaded methods MUST change the argument list.
- Overloaded methods CAN change the return type.
- Overloaded methods CAN change the access modifier.
- Overloaded methods CAN declare new or broader checked exceptions.

Exam Watch

exam Watch

Be careful to recognize when a method is overloaded rather than overridden. You might see a method that appears to be violating a rule for overriding, but that is actually a legal overload, as follows:

```
public class Foo {
   public void doStuff(int y, String s) { }
   public void moreThings(int x) { }
}
class Bar extends Foo {
   public void doStuff(int y, long s) throws IOException { }
}
```

It's tempting to see the `IOException` as the problem, because the overridden `doStuff()` method doesn't declare an exception, and `IOException` is checked by the compiler. But the `doStuff()` method is not overridden! Subclass `Bar` overloads the `doStuff()` method, by varying the argument list, so the `IOException` is fine.

Exam Watch

exam Watch

Don't be fooled by a method that's overloaded but not overridden by a subclass. It's perfectly legal to do the following:

```
public class Foo {
   void doStuff() { }
}
class Bar extends Foo {
   void doStuff(String s) { }
}
```

The Bar class has two `doStuff()` methods: the no-arg version it inherits from Foo (and does not override), and the overloaded `doStuff(String s)` defined in the Bar class. Code with a reference to a Foo can invoke only the no-arg version, but code with a reference to a Bar can invoke either of the overloaded versions.

Constructing the Super Classes

- Subclasses must construct the super class
  - automatic if the super class has a default constructor
  - otherwise, must use `super()` to call a specific constructor
  - `super()` must be the first statement in the constructor

```
public class Lion extends Animal{
      private Color maneColour;

      public Lion(Color c, int sz, Color mane){
            super(c,sz);
            this.maneColour=mane;
      }

      public Lion(Color c, int sz){
            this(c,sz,Color.BLACK);
      }
}
```

More Visibility Modifiers

- Control class, method and field visibility
  - two already seen
  - `private` and `public`
- New modifier used with subclasses
  - `protected` accessible to subclasses and classes in the same package

Object Hierarchy

- All Java classes are ultimately derived from `Object`
  - by default a class extends `java.lang.Object`
  - sole exception: `Object` itself has no parent class
- This forces all classes to be part of one hierarchy
  - `Object` at the top of giant tree structure
- Due to the principle of Substitutability
  - all java objects can be treated at a high level of abstraction
- Allows support for common language features to be built into all objects
  - multithreading (`wait()`, `notifyAll()`)
  - diagnostics (`toString()`)

The Object Class

`boolean equals(Object)`	Compare two objects for equality
`String toString()`	String representation
`int hashCode()`	Used with Collections
`Class getClass()`	Return Class of object
`void notify()`	Support for Multithreading
`void notifyAll()`	Support for Multithreading
`void wait()`	Support for Multithreading
`Object clone()`	Create a bitwise copy of object

Object Reference Type Casting

- Object references can be cast up or down the hierarchy
  - type casting does not change the object but tells the compiler how the object is used (strong compile time checking)

```
private Object[] getData(){
   Object[] ans=new Object[3];
   ans[0]="This is the answer";
   ans[1]=new Integer(42);
   ans[2]=new int[]{1,2,3};
   return ans;
}
```

```
Object[] list=getData();

for(int i=0;i<list.length;i++){
   Object obj=list[i];
   if(obj instanceof Integer){
   int n=((Integer)obj).intValue();
   }
   else if(obj instanceof int[]){
         int[] array = (int[])obj;
         for(int j=0;j<array.length;i++){
         System.out.println("Array element "+j+" = "+array[j]);
         }
      }
   System.out.println("String value = "+obj);
   System.out.println("Class = "+obj.getClass());
}
```

`final` Modifier

- The `final` modifier on classes
  - prevents the class from being extended
  - can improve compiler and runtime optimisation
- The `final` modifier on methods
  - prevents a method being overridden in derived classes
- The `final` modifier on variables
  - makes the “variable” constant
  - fixes the value of a variable for its lifetime
  - must be supplied with an initial value
- Putting final binds it at compile time and gives a little performance boost

```
final static int MAX_PLAYERS = 10
```

Abstract Classes

- Abstraction is the distilling of essential features
  - eliminates any non-essential, superfluous detail
  - concentrate on what can be done, rather than how it is done
- Abstract classes are a class “skeleton”
  - cannot instantiate objects from an abstract class
  - concrete classes are classes which can be instantiated
- Abstract classes are often partially implemented
  - a class is abstract if one of its methods is defined as `abstract`
  - abstract methods are given as prototypes with no code body
- Subclasses must override (i.e. implement) all abstract methods in the superclass, or they are too abstract

```
public abstract class Animal{
      public abstract void eat();
}
```

Example

```
public abstract class Animal{
      public abstract void eat();
}
```

```
public class Lion extends Animal{

   @Override
   public void eat() {
      System.out.println("Give me some flesh");
   }
}
```

```
public void feed{Animal a){
 a.eat();
}
...
Animal l=new Lion();
Animal g=new Giraffe();
Animal s = new Seal();
...

feed(l);feed(g);feed(s);
```

```
public class Giraffe extends Animal{

   @Override
   public void eat() {
      System.out.println("I am vegetarian");
   }
}
```

```
public class Seal extends Animal{

   @Override
   public void eat() {
      System.out.println("Give me some fish");
   }
}
```

Interfaces

- Interfaces: abstract types with no implementation
  - all methods are implicitly `public`
  - all methods are implicitly `abstract` – no default behavior
  - any variables are implicitly `static final` – must be initialised
- Unlike abstract classes, interfaces reside in a separate inheritance hierarchy
  - classes can implement multiple interfaces – multiple inheritance
  - interfaces can be extended – interface inheritance

```
public interface Flyable{
      void fly();
}
```

```
public class Aircraft implements Flyable{

      @Override
      public void fly() {
            ...
      }
}
```

Programming to an Interface

- Implementing an interface is implementing a contract
- Interface inheritance can be used to implement “a kind of” relationship
- Useful to produce “substitutable” classes
- Enables Java to implement polymorphic behavior
- Java API Example
  - Collections framework
  - JDBC framework for database access
  - Most “plug-in” architectures (e.g. Applets, servlets, ...)

Question

1. Given:

```
public abstract interface Frobnicate { public void twiddle(String s); }
```

Which is a correct class? (Choose all that apply.)

A.
```
public abstract class Frob implements Frobnicate {
    public abstract void twiddle(String s) { }
}
```

B.
```
public abstract class Frob implements Frobnicate { }
```

C.
```
public class Frob extends Frobnicate {
    public void twiddle(Integer i) { }
}
```

D.
```
public class Frob implements Frobnicate {
    public void twiddle(Integer i) { }
}
```

E.
```
public class Frob implements Frobnicate {
    public void twiddle(String i) { }
    public void twiddle(Integer s) { }
}
```

Answer

Answer:

☑ B is correct, an `abstract` class need not implement any or all of an interface's methods. E is correct, the class implements the interface method and additionally overloads the `twiddle()` method.

☒ A is incorrect because `abstract` methods have no body. C is incorrect because classes implement interfaces they don't extend them. D is incorrect because overloading a method is not implementing it.

Question

Given:

```
class Clidder {
  private final void flipper() { System.out.println("Clidder"); }
}

public class Clidlet extends Clidder {
  public final void flipper() { System.out.println("Clidlet");  }
  public static void main(String [] args) {
    new Clidlet().flipper();
} }
```

What is the result?

A. `Clidlet`

B. `Clidder`

C. `Clidder`
`Clidlet`

D. `Clidlet`
`Clidder`

E. Compilation fails

Answer

Answer:

☑ A is correct. Although a final method cannot be overridden, in this case, the method is private, and therefore hidden. The effect is that a new, accessible, method flipper is created. Therefore, no polymorphism occurs in this example, the method invoked is simply that of the child class, and no error occurs.

☒ B, C, D, and E are incorrect based on the preceding. (Objective 5.3)

Question

Given the following,

```
1. class X { void do1() { } }
2. class Y extends X { void do2() { } }
3.
4. class Chrome {
5.   public static void main(String [] args) {
6.     X x1 = new X();
7.     X x2 = new Y();
8.     Y y1 = new Y();
9.     // insert code here
10. } }
```

Which, inserted at line 9, will compile? (Choose all that apply.)

A. `x2.do2();`

B. `(Y)x2.do2();`

C. `((Y)x2).do2();`

D. None of the above statements will compile

Answer

Answer:

☑ C is correct. Before you can invoke Y's `do2` method you have to cast `x2` to be of type `Y`. Statement B looks like a proper cast but without the second set of parentheses, the compiler thinks it's an incomplete statement.

☒ A, B and D are incorrect based on the preceding.

Question

Given:

```
3. class Dog {
4.   public void bark() { System.out.print("woof "); }
5. }
6. class Hound extends Dog {
7.   public void sniff() { System.out.print("sniff "); }
8.   public void bark() { System.out.print("howl "); }
9. }
10. public class DogShow {
11.   public static void main(String[] args) { new DogShow().go(); }
12.   void go() {
13.     new Hound().bark();
14.     ((Dog) new Hound()).bark();
15.     ((Dog) new Hound()).sniff();
16.   }
17. }
```

What is the result? (Choose all that apply.)

A. `howl howl sniff`

B. `howl woof sniff`

C. `howl howl` followed by an exception

D. `howl woof` followed by an exception

E. Compilation fails with an error at line 14

F. Compilation fails with an error at line 15

Answer

Answer:

☑ F is correct. Class `Dog` doesn't have a `sniff` method.

☒ A, B, C, D, and E are incorrect based on the above information.

Question

Given:

```
 3. public class Redwood extends Tree {
 4.   public static void main(String[] args) {
 5.     new Redwood().go();
 6.   }
 7.   void go() {
 8.     go2(new Tree(), new Redwood());
 9.     go2((Redwood) new Tree(), new Redwood());
10.   }
11.   void go2(Tree t1, Redwood r1) {
12.      Redwood r2 = (Redwood)t1;
13.      Tree t2 = (Tree)r1;
14.   }
15. }
16. class Tree { }
```

What is the result? (Choose all that apply.)

A. An exception is thrown at runtime
B. The code compiles and runs with no output
C. Compilation fails with an error at line 8
D. Compilation fails with an error at line 9
E. Compilation fails with an error at line 12
F. Compilation fails with an error at line 13

Answer

Answer:

☑ A is correct, a ClassCastException will be thrown when the code attempts to downcast a Tree to a Redwood.

☒ B, C, D, E, and F are incorrect based on the above information.

Question

Given:

```
3. class Mammal {
4.   String name = "furry ";
5.   String makeNoise() { return "generic noise"; }
6. }
7. class Zebra extends Mammal {
8.   String name = "stripes ";
9.   String makeNoise() { return "bray"; }
10. }
11. public class ZooKeeper {
12.   public static void main(String[] args) { new ZooKeeper().go(); }
13.   void go() {
14.     Mammal m = new Zebra();
15.     System.out.println(m.name + m.makeNoise());
16.   }
17. }
```

What is the result?

A. `furry bray`

B. `stripes bray`

C. `furry generic noise`

D. `stripes generic noise`

E. Compilation fails

F. An exception is thrown at runtime

Answer

Answer:

☑ A is correct. Polymorphism is only for instance methods.

☒ B, C, D, E, and F are incorrect based on the above information. (Objectives 1.5, 5.4)

Contact Info

- trainers@finaldesk.com
- rishabh@finaldesk.com
- nilesh@finaldesk.com
- jignesh@finaldesk.com
- yash@finaldesk.com
- anand@finaldesk.com