Announcement

- Final exam:
  - Thursday, April 22nd at 9:00am (until 12:00) at the GYM
  - Covers everything, but with higher emphasis on OOP (don't forget recursion, arrays, exceptions and data-structures.)
  - Structure: 12 Multiple Choice Questions, 5 Short Problems, 2 Programming problems
- Assignment 6 posted this afternoon
- Last day of lectures: Tuesday, April 12 at the same time and place.

Polymorphism

- Polymorphism is a tool that permits abstraction and reusability
- A polymorphic method is a method which can receive as input any object whose class is a subclass of the methods's parameter.
- Ad-hoc polymorphism is overloading (providing separate methods for each expected parameter type)
- Parametric polymorphism relies on dynamic-dispatching. Dynamic-dispatching is the process by which the run-time system directs the message of an object to the appropriate subclass.
- A dynamic-dispatch can be decided only at run-time, not at compile-time, because the compiler cannot know which is the actual object passed as argument to a polymorphic method. Furthermore, the same method might be called with different objects from different classes during the execution of the program.

Polymorphism

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
}
class Human extends Creature {
  void move()
  {
    System.out.println("Walking...");
  }
}
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
}
```

Ad-hoc Polymorphism (Overloading)

```
class Zoo {
  void animate(Human h)
  {
    h.move();
  }
  void animate(Martian m)
  {
    m.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick);  // Polymorphic call
  }
}
```

Ad-hoc Polymorphism (Overloading)

```
class Penguin extends Creature {
  void stumble()
  {
    System.out.println(“Ouch”);
  }
}

class Zoo {
  void animate(Human h)
  {
    h.move();
  }
  void animate(Martian m)
  {
    m.move();
  }
  void animate(Penguin p)
  {
    p.move();
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move();
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human yannick = new Human();
    Martian ernesto = new Martian();
    my_zoo.animate(ernesto); // Polymorphic call
    my_zoo.animate(yannick);  // Polymorphic call
  }
}
```

Checking the type of a reference

- To find out whether a reference `r` is an instance of a particular class `C` we use the boolean expression:

```
r instanceof C
```

- This is normally used whenever we do casting:

```
class Human extends Creature {
  void move()
  {
    System.out.println(“Walking...”);
  }
  void jump()
  {
      System.out.println("Up and down");
  }
}
```

Checking the type of a reference

```
class Martian extends Creature {
  void move()
  {
    System.out.println("Crawling...");
  }
  void hop()
  {
      System.out.println("Down and to the left");
  }
}
class Zoo {
  void animate(Creature c)
  {
    if (c instanceof Human)
      ((Human)c).jump();
    else if (c instanceof Martian)
      ((Martian)c).hop();
    c.move();
  }
}
```

Controling dynamic-dispatch with casting

```
class FlyingMartian extends Martian {
  void move()
  {
    System.out.println(“Gliding...”);
  }
}

class ZooTest {
  public static void main(String[] args)
  {
    FlyingMartian peng = new FlyingMartian();
    peng.move();
    ((Martian)peng).move();
    ((Creature)peng).move();
    ((Human)peng).move(); // Error peng is not Huma
  }
}
```

Abstract classes

- A class with default behaviour:

```
class Creature {
  boolean alive;
  void move()
  {
    System.out.println("Here we go...");
  }
}
```

- An abstract class: subclasses must provide implementation

```
abstract class Creature {
  boolean alive;
  abstract void move();
}
```

Abstract classes

- An abstract class is a class that has at least one abstract method
- An abstract method is a method which is not implemented (i.e. has no body) and must be overriden (i.e. must implemented by the subclasses.)
- An abstract class is used to represent an abstract concept which captures the common structure and behaviour of several classes, but leaves some detail to the subclasses.
- Abstract classes force the use of parametric polymorphism.

Abstract classes

- There cannot be instances of abstract classes.

```
Creature kowe = new Creature(); // Wrong!
//because
kowe.move(); // What would be executed here?
```

- The abstract methods *must* be implemented in the subclasses of an abstract class (unless the subclass itself is also abstract.) This is, there is no default behaviour for an abstract method.

Abstract classes

- An abstract class can have non-abstract methods (which usually represent the "default behaviour" of a method:)

```
abstract class Creature
{
  boolean alive, hungry;
  abstract void move();
  void eat()
  {
    System.out.println("Hmmm...");
    hungry = false;
  }
}
```

Interfaces

- Interfaces are (equivalent to) purely abstract classes, i.e. classes where all the methods are abstract

```
interface Creature
{
  void move();
  void eat();
}
```

is the same as

```
abstract class Creature
{
  abstract void move();
  abstract void eat();
}
```

Interfaces

```
class Human implements Creature
{
  void move()
  {
    System.out.println("I'm walking...");
  }
  void eat()
  {
    System.out.println("I'm eating...");
  }
  void jump()
  {
    System.out.println("Up and down...");
  }
}
```

Using interfaces for generalization

```
class CDPlayer {
  int song;
  boolean stopped;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void play() { stopped = false; }
  void ff() { song++; }
  void pause() { stopped = true; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Using interfaces for generalization

```
class TapeRecorder {
  boolean stopped, recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces

```
interface MusicPlayer {
  void play();
  void ff();
  void pause();
  void stop();
}
```

Interfaces

```
class CDPlayer implements MusicPlayer {
  int song;
  boolean stopped;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void play() { stopped = false; }
  void ff() { song++; }
  void pause() { stopped = true; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Interfaces

```
class TapeRecorder implements MusicPlayer {
  boolean stopped, recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces

```
class PlayerTest {
  static void test(MusicPlayer p)
  {
    p.play();
    p.ff();
    p.pause();
    p.play();
    if (p instanceof TapeRecorder) {
      ((TapeRecorder)p).record(new Tape());
    }
    p.stop();
  }
}
```

Interfaces

```
class SoundStudio {
  public static void main(String[] args)
  {
    MusicPlayer[] players = { new CDPlayer(),
                              new TapeRecorder(),
                              new CDPlayer() };
    for (int i = 0; i < players.length; i++) {
      PlayerTest.test(players[i]);
        // polymorphic call.
    }
  }
}
```

Abstract classes

```
abstract class MusicPlayer {
  boolean stopped;
  void play() { stopped = false; }
  void ff() { }
  void pause() { stopped = true; }
  abstract void stop();
}
```

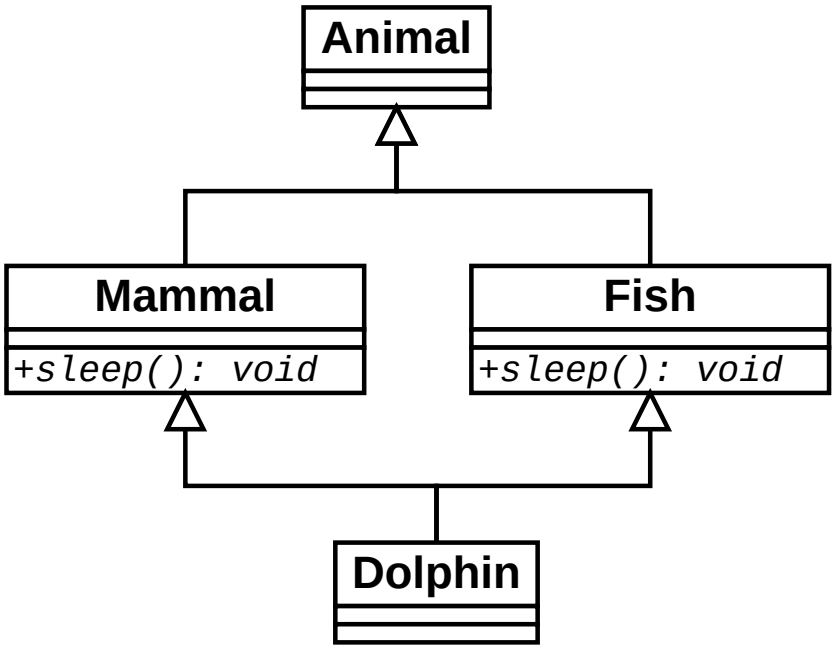
Abstract classes

```
class CDPlayer extends MusicPlayer {
  int song;
  CDPlayer()
  {
    stopped = true;
    song = 0;
  }
  void ff() { song++; }
  void stop()
  {
    stopped = true;
    song = 0;
  }
}
```

Abstract classes

```
class TapeRecorder extends MusicPlayer {
  boolean recording;
  Tape t;
  TapeRecorder() {
    stopped = true;
    recording = false;
    t = null;
  }
  void ff() { }
  void stop() {
    stopped = true;
    recording = false;
  }
  void record(Tape x) {
    recording = true;
    t = x.clone();
  }
}
```

Interfaces and multiple inheritance




```
class A extends B, C { ... } // Error
class A implements B, C { ... } // OK
```

- Java does not support multiple inheritance of classes, but any class can implement more than one interface.

Interfaces

- Multiple inheritance is supported for interfaces

```
class A extends B implements C, D, E { ... }
```

- ...because the methods in the interfaces are abstract, which means that they must be implemented in A, so there is no ambiguity problem when calling a method.

Generic programming

- A generic function/procedure/method is one whose algorithm doesn't depend on the types of its arguments
- Generic procedures are abstract, and therefore highly reusable
- In java generic procedures are implemented using parametric polymorphism
- Example generic sorting:
  - To sort an array of objects, where the objects have a key, and keys are comparable.

```
interface Comparable
{
  public int compareTo(Comparable obj);
}
```

Generic programming

```
class Student extends Human implements Comparable
{
  private String name;
  private long id;
  private int age;
  //...
  public int compareTo(Student s)
  {
    if (this.age < s.age) return -1;
    else if (this.age > s.age) return 1;
    if (this.name.compareTo(s.name) < 0)
      return -1;
    else if (this.name.compareTo(s.name) > 0)
      return 1;
    return 0;
  }
}
```

Generic programming

```
class SortAlgorithms {
  static void insertion_sort(Comparable[] a)
  {
    int i, j;
    Comparable key;
    for (j = 1; j < a.length; j++) {
      key = a[j];
      i = j - 1;
      while (i >= 0
        && key.compareTo(a[i]) < 0 ) {
        a[i+1] = a[i];
        i--;
      }
      a[i+1] = key;
    }
  }
}
```

Generic programming

```
static void insertion_sort(Book[] a)
{
  int i, j;
  String key;
  for (j = 1; j < a.length; j++) {
    key = a[j].get_title();
    i = j - 1;
    while (i >= 0
      && key.compareTo(a[i].get_title()) < 0 ) {
      a[i+1] = a[i];
      i--;
    }
    a[i+1] = key;
  }
}
```

Generic programming

```
class GenericSortTest {
  public static void main(String[] args)
  {
    Student[] course = new Student[230];
    enter_info(course);
    SortingAlgorithms.insertion_sort(course);
    String[] words = {“one”,“two”,“three”,“four”};
    SortingAlgorithms.insertion_sort(words);
  }
  static void enter_info(Student[] course)
  {
    for (int i = 0; i < course.length; i++) {
      String name = Keyboard.readString();
      int age = Keyboard.readInt();
      long id = Keyboard.readLong();
      course[i] = new Student(name, age, id);
    }
  }
}
```

Changing visibility in subclasses

- A public method cannot be overriden by a private or protected method:

```
class A {
    public void m()
    {
        System.out.println("A");
    }
}
class B extends A {
    private void m()
    {
        System.out.println("B");
    }
}
```

Changing visibility in subclasses

- A method can be overriden by method with weaker access priviledges:

```
class A {
    protected void m()
    {
        System.out.println("A");
    }
}
class B extends A {
    public void m()
    {
        System.out.println("B");
    }
}
```

Object

- Object is a class in the standard Java library which is a superclass to all.

- It contains methods

```
public boolean equals(Object o)
protected Object clone()
public String toString()
public Class getClass()
```

- A method whose argument is of type Object can receive any object from any class as argument. (maximum possible polymorphism.)

- Whenever an object appears in a String expression, the method toString is invoked automatically

Object

```
class Human {
  String name;
  public String toString()
  {
    return “My name is “+name;
  }
}
class Test {
  public static void main(String[] args)
  {
    Human h = new Human();
    h.name = “Kelly”;
    String s = “”+h;
    // Same as String s = “”+h.toString();
  }
}
```

Using the Object class

```
import java.util.Vector;
class Test {
  void p() {
    Vector v = new Vector();
    v.addElement(new Integer(2));
    v.addElement(new Integer(5));
    v.insertElementAt(new Integer(3), 1);
    Integer i = (Integer)v.elementAt(2);
    int n = i.intValue();
  }
}
```

Remarks on constructors

```
class A {
    String s;
    A(String q)
    {
        s = "hello "+q;
    }
}

public class ConsTest {
    public static void main(String[] args)
    {
        A x = new A();  // Error
        System.out.println(x.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A(String q)
    {
        s = "hello "+q;
    }
}

public class ConsTest {
    public static void main(String[] args)
    {
        A x = new A(“bye”);
        System.out.println(x.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A() { s = "bonjour "; }
    A(String q)
    {
        s = "hello "+q;
    }
}

public class ConsTest {
    public static void main(String[] args)
    {
        A x = new A();
        System.out.println(x.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A()
    {
        s = "hello ";
    }
}
class B extends A {
    int n;
}
public class ConsTest {
    public static void main(String[] args)
    {
        B b1 = new B();
        System.out.println(b1.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A(String q)
    {
        s = "ask "+q;
    }
}
class B extends A {
    int n;
}
public class ConsTest
{
    public static void main(String[] args)
    {
        B b1 = new B();
        System.out.println(b1.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A() { s = "hello "; }
}

class B extends A {
    int n;
    B(int i)
    {
        n = i;
    }
}

public class ConsTest {
    public static void main(String[] args)
    {
        B b1 = new B(5);
        System.out.println(b1.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A(String q) { s = "hello "+q; }
}
class B extends A {
    int n;
    B(int i)
    {   // Error: no A()
        n = i;
    }
}
public class ConsTest {
    public static void main(String[] args)
    {
        B b1 = new B(5);
        System.out.println(b1.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A(String q) { s = "hello "+q; }
}
class B extends A {
    int n;
    B(int i)
    {
        super(“bye”);
        n = i;
    }
}
public class ConsTest {
    public static void main(String[] args)
    {
        B b1 = new B(5);
        System.out.println(b1.s);
    }
}
```

Remarks on constructors

```
class A {
    String s;
    A()  { s = "bye "; }
    A(String q) { s = "hello "+q; }
}
class B extends A {
    int n;
    B(int i)
    {
        super("salut");
        n = i;
    }
}
public class ConsTest {
    public static void main(String[] args)
    {
        B b1 = new B(5);
        System.out.println(b1.s);
    }
}
```

Remarks on constructors

- If a class `A` does not have a constructor, then it implicitely has a default constructor with no parameters

  ```
  A() { super(); }
  ```

- If a class `A` has a constructor with parameters, then it does not implicitely have a default constructor `A()`

- Constructors are not inherited

- All constructors have an implicit call to the superclass's default constructor, unless it explicitely calls a non-default constructor from the parent.

Generic programming

```
interface Indexed {
  public Comparable get_key();
}

class Student extends Human implements Indexed {
  private long id;
  private StudentKey key;
  public Student(String name, int age, long id)
  {
    key = new StudentKey(name, age);
    this.id = id;
  }
  //...
  public StudentKey get_key()
  {
    return key;
  }
}
```

Generic programming

```
class StudentKey implements Comparable {
  private String name;
  private int age;
  public StudentKey(String n, int a)
  {
    name = n;
    age = a;
  }
  public int compareTo(StudentKey s)
  {
    if (this.age < s.age) return -1;
    else if (this.age > s.age) return 1;
    if (this.name.compareTo(s.name) < 0)
      return -1;
    else if (this.name.compareTo(s.name) > 0)
      return 1;
    return 0;
  }
}
```

Generic programming

```
class SortAlgorithms {
  static void insertion_sort(Indexed[] a)
  {
    int i, j;
    Comparable key;
    for (j = 1; j < a.length; j++) {
      key = a[j].get_key();
      i = j - 1;
      while (i >= 0
        && key.compareTo(a[i].get_key()) < 0 ) {
        a[i+1] = a[i];
        i--;
      }
      a[i+1] = key;
    }
  }
}
```

The end