Announcements

- Final exam: April 20th at 9:00am at the GYM
- Assignment 6, to be posted soon

Polymorphism

- Polymorphism means "many forms."
- Polymorphism is the characteristic of being able to assign a different meaning or usage to something in different contexts
- If a class A has a method m we could give different meaning to m by defining subclasses that override m, and therefore the result of executing m depends on the context, since the context decides which subclass is instantiated.

Polymorphism

```
class A
{
  void m() { ... }
}

class B extends A
{
  void m() { ... }
}

class C extends A
{
  void m() { ... }
}
```

Polymorphism

```
class D
{
  void m()
  {
    A obj;
    obj = new B();
    obj.m();
    obj = new C();
    obj.m();
  }
}
```

Polymorphism

```
class D
{
  void q(A obj)
  {
    obj.m();
  }
  void p()
  {
    A obj;
    obj = new B();
    q(obj);
    obj = new C();
    q(obj);
  }
}
```

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...");
  }
}
```

Polymorphism

```
public class ZooTest
{
  public static void main(String[] args)
  {
    Human yannick = new Human();
    Martian ernesto = new Martian();
    ernesto.move();
    yannick.move();
  }
}
```

Polymorphism

- A polymorphic method is a method which can accept more than one type of argument
- Kinds of polymorphism:
  - Overloading (Ad-hoc polymorphism)
  - Parametric polymorphism

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();
    Penguin paco = new Penguin();
    my_zoo.animate(ernesto);
    my_zoo.animate(yannick);
    my_zoo.animate(paco);
  }
}
```

Polymorphism

```
class Creature
{
  boolean alive;
  void move()
  {
    System.out.println("The way I move is by...");
  }
  void dance()
  {
    System.out.println("Look, I'm dancing...");
    move();  // Dynamic-dispatch
  }
}
```

Parametric Polymorphism

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

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);
    my_zoo.animate(yannick);
  }
}
```

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

- 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;
  abstract void move();
  void dance()
  {
    System.out.println("Look, I'm dancing...");
    move();
  }
}
```

Interfaces

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

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

is the same as

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

Interfaces

```
class Human implements Creature
{
  void move()
  {
    System.out.println(“I’m walking...”);
  }
  void dance()
  {
    System.out.println(“1-2-3-spin-1-2-3-step...”)
  }
  void jump()
  {
    System.out.println(“Up and down...”);
  }
}
```

Runtime type-checking

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

Runtime type-checking

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

Runtime type-checking

```
class Martian extends Creature
{
  void move()
  {
    System.out.println("Crawling...");
  }
  void hop()
  {
      System.out.println("Down and to the left");
  }
}
```

Runtime type-checking

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

Runtime type-checking

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

Runtime type-checking

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    c.jump();  // ERROR!
    c.hop();   // ERROR!
  }
}
```

Runtime type-checking

```
class Zoo
{
  void animate(Human c)
  {
    c.move();
    c.jump();
  }
  void animate(Martian c)
  {
    c.move();
    c.hop();
  }
}
```

Runtime type-checking

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c is a Human)  // Not legal Java
    {
      c.jump();
    }
    if (c is a Martian) // Not legal Java
    {
      c.hop();
    }
  }
}
```

Runtime type-checking

- Most type information is checked at compile-time
- But the exact type of a reference is only known at run-time
- To find out whether a reference `r` is an instance of a particular class `C` we use the boolean expression:

```
r instanceof C
```

Runtime type-checking

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c instanceof Human)
    {
      c.jump();
    }
    if (c instanceof Martian)
    {
      c.hop();
    }
  }
}
```

Runtime type-checking

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c instanceof Human)
    {
      c.jump(); // ERROR!: Creature cannot jump
    }
    if (c instanceof Martian)
    {
      c.hop();  // ERROR!: Creature cannot hop
    }
  }
}
```

Casting object references

- Casting is like putting a special lens on an object
- A casting expression is of the form

  ```
  (type) expr
  ```

  where `type` is any type (primitive or user-defined) and `expr` is an expression which evaluates to an object reference whose type is compatible with `type`.

- For example:

  ```
  (int)5.4
  (double)3
  (int)(2.0 / 3)
  ((double) 2) / 3
  ```

Casting object references

- Classes are types
- We can do casting of object references:

```
(Creature)ernesto
(Martian)zork
```

- Not all casts are possible

```
(int) "Hello"
(Engine) yannick
```

Casting

- If a variable is a reference of type A, it can be assigned any object whose type is a subclass of B.

```
Human greg = new Human();
Creature c = greg;
```

- But a reference of type B cannot be assigned directly reference of type A, if B is a subclass of A (because A has less attributes than required by B):

```
Creature d = new Creature();
Martian m = d;          // Error
```

Casting object references

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

Casting object references

```
class Martian extends Creature
{
  void move()
  {
    System.out.println("Crawling...");
  }
  void hop()
  {
      System.out.println("Down and to the left");
  }
}
```

Casting object references

```
public class ZooTest {
  public static void main(String[] args)
  {
    Creature ernesto = new Creature();
    Martian zork = ernesto;
    zork.hop();
  }
}
```

Casting object references

```
public class ZooTest {
  public static void main(String[] args)
  {
    Creature ernesto = new Creature();
    Martian zork = ernesto;
    zork.hop();  // Error! ernesto cannot hop
  }
}
```

Casting object references

- ...however, if we know that a reference x of type A points to an object of type B (and B is a subclass of A,) then we can force to see x as being of type B by using a casting expression:

```
Creature e = new Martian();
Martian f = (Martian)e;
```

Casting object references

```
public class ZooTest {
  public static void main(String[] args)
  {
    Creature ernesto = new Martian();
    Martian zork = (Martian)ernesto;
    zork.hop();  // OK: ernesto can hop
  }
}
```

Casting object references

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c instanceof Human)
    {
      c.jump(); // ERROR!: Creature cannot jump
    }
    if (c instanceof Martian)
    {
      c.hop();  // ERROR!: Creature cannot hop
    }
  }
}
```

Casting object references

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c instanceof Human)
    {
      Human h = (Human)c;
      h.jump(); // OK
    }
    if (c instanceof Martian)
    {
      Martian m = (Martian)c;
      m.hop();  // OK
    }
  }
}
```

Casting object references

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    if (c instanceof Human)
    {
      ((Human)c).jump(); // OK
    }
    if (c instanceof Martian)
    {
      ((Martian)c).hop();  // OK
    }
  }
}
```

Casting object references

```
class Zoo
{
  void animate(Creature c)
  {
    c.move();
    ((Human)c).jump(); // Bad idea...
    ((Martian)c).hop();  // Bad idea...
  }
}
```

Casting object references

```
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
  }
}
```

Narrowing and Widening casts

- Suppose class A has B as a subclass.
- Narrowing casts make a reference to a B object into an A object

```
B z = new B();
A w = (A)z; // Narrowing; Same as A w = z;
```

- Widening casts make a reference to an A object into a B object

```
A x = new B();  // Narrowing
B y = (B)x;  // Widening
```

- Sometimes it is necessary to make an explicit narrowing conversion if we want to force an object to behave as one of its ancestors, for example to access some overriden method.

Narrowing and Widening casts

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

Narrowing and Widening casts

```
class ZooTest
{
  public static void main(String[] args)
  {
    FlyingMartian peng = new FlyingMartian();
    peng.move();
    ((Martian)peng).move();
    ((Creature)peng).move();
  }
}
```

Narrowing and Widening casts

```
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
  }
}
```

Casting

- Casting is used to access attributes or methods that would otherwise be inaccessible.

- Case 1: accessing attributes and methods that have been overriden
  - we use `super` if we are trying to access them from within the same class,
  - but we use casting when trying to access them from a different class

Casting

```
class A {
  int x = 17;
}
class B extends A {
  int x = 29;
  void p()
  {
    System.out.print(x);
  }
}
```

Casting

```
class A {
  int x = 17;
}
class B extends A {
  int x = 29;
  void p()
  {
    System.out.print(super.x);
  }
}
```

Casting

```
class C {
  void q()
  {
    B b;
    b = new B();
    System.out.print(b.x);
  }
}
```

Casting

```
class C {
  void q()
  {
    B b;
    b = new B();
    System.out.print(b.super.x); // Wrong!
  }
}
```

Casting

```
class C {
  void q()
  {
    B b;
    b = new B();
    A a;
    a = b;
    System.out.print(a.x);
  }
}
```

Casting

```
class C {
  void q()
  {
    B b;
    b = new B();
    System.out.print(((A)b).x);
  }
}
```

Casting

```
class A {
  void m()
  {
    System.out.print(17);
  }
}
class B extends A {
  void m()
  {
    System.out.print(29);
  }
}
```

Casting

```
class C {
  void q()
  {
    B b;
    b = new B();
    b.m();
    A a;
    a = b;
    a.m();
  }
}
```

Casting

```
class C {
  void q()
  {
    B b;
    b = new B();
    b.m();
    ((A)b).m();
  }
}
```

Casting

- Casting is used to access attributes or methods that would otherwise be inaccessible.

- Case 2: accessing attributes and methods in a polymorphic method
  - a variable `x` of type `A` can be assigned an instance of any subclass `B` of `A`.
  - to access attributes or methods from the subclass `B`, we cast `x`

Casting

```
class A {
  void m()
  {
    System.out.print(17);
  }
}
class B extends A {
  void m()
  {
    System.out.print(29);
  }
  void p()
  {
    System.out.print(3);
  }
}
```

Casting

```
class C {
  void q(A x)
  {
    x.m();
  }
  void r()
  {
    B b = new B();
    q(b);
  }
}
```

Casting

```
class C {
  void q(A x)
  {
    x.m();
    x.p();
  }
  void r()
  {
    B b = new B();
    q(b);
  }
}
```

Casting

```
class C {
  void q(A x)
  {
    x.m();
    ((B)x).p();
  }
  void r()
  {
    B b = new B();
    q(b);
  }
}
```

Casting

```
class C {
  void q(A x)
  {
    x.m();
    if (x instanceof B) {
      ((B)x).p();
    }
  }
  void r()
  {
    B b = new B();
    q(b);
  }
}
```

The end