Casting and instanceof

- Casting is like putting a special lens on an object
- Casting is used to access attributes or methods that would otherwise be inaccessible
- 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`.
- Not all casts are possible

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

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();
  }
  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);
  }
}
```

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
```

- ...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

```
class Creature {
  boolean alive;
  // ...
}
class Martian extends Creature {
  int legs, wings;
  // ...
}

// Somewhere else...
Creature d = new Creature();
Martian m = d;
int k = m.legs + m.wings; // Error!
// ...because d does not have legs or wings
```

Casting

```
class Creature {
  boolean alive;
  void move() { ... }
}
class Martian extends Creature {
  int legs, wings;
  void move() { ... }
  void hop() { ... }
}

// Somewhere else...
Creature d = new Creature();
Martian m = d;
m.hop(); // Error!
// ...because d cannot hop
```

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();
  }
}
```

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();
  }
}
```

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();
  }
}
```

The end