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 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();
  }
}
```

Ad-hoc Polymorphism (Overloading)

```
public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human joseph = new Human();
    Martian ernesto = new Martian();
    Penguin paco = new Penguin();

    my_zoo.animate(ernesto);
      // calls move from Martian
    my_zoo.animate(joseph);
      // calls move from Human
    my_zoo.animate(paco);
      // calls move from Creature
  }
}
```

Parametric Polymorphism

```
class Zoo {
  void animate(Creature c)
  {
    c.move(); // Dynamic-dispatch
    // move *must* be defined in class Creature
  }
}

public class ZooTest {
  public static void main(String[] args)
  {
    Zoo my_zoo = new Zoo();
    Human joseph = new Human();
    Martian ernesto = new Martian();
    Penguin paco = new Penguin();

    my_zoo.animate(ernesto);
    my_zoo.animate(joseph);
    my_zoo.animate(paco);
  }
}
```

Accessing super

```
class Human extends Creature {
  void move()
  {
    super.move();
    System.out.println(“Walking...”);
  }
}
class Martian extends Creature {
  void move()
  {
    super.move()
    System.out.println(“Crawling...”);
  }
}
class Penguin extends Creature {
  void stumble()
  {
    super.move();
    System.out.println(“Ouch”);
  }
}
```

Casting and instanceof

- Casting is like putting a special lens on an object, in order to observe the object as if it was of a different type.

- 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) joseph
```

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;  // Wrong!
```

Casting

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

// Somewhere else...
Creature d = new Creature();
Martian m = d;
int n = 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
```

Casting

- ...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;
int n = f.legs * f.wings;
((Martian)e).hop(); // same as f.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();
    else if (c instanceof Penguin)
      ((Penguin)c).stumble();
    c.move();
  }
}
```

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...”);
  }
}

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
  }
}
```

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 eny 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();
  }
}
```

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;
  }
}
```