Victoria University
of Wellington, New Zealand
Te Whare Wananga o te
Upoko o te Ika a Maui
Aotearoa

SWEN221 Software Development

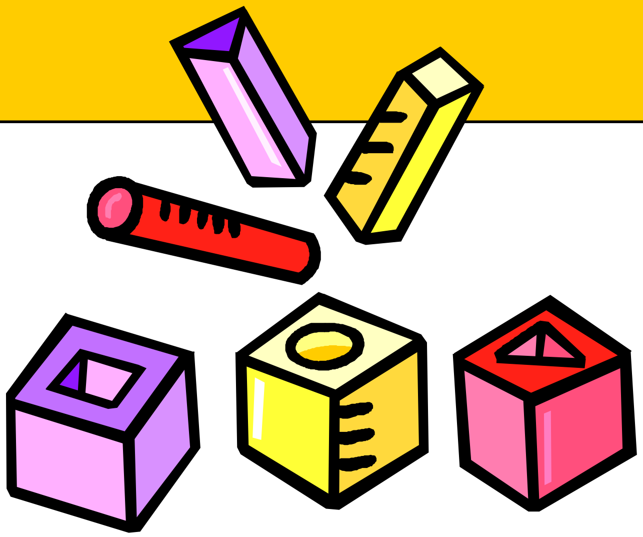
Polymorphism

Thomas Kuehne

Victoria University

(slides modified from slides by David J. Pearce & Nicholas Cameron & James Noble & Petra Malik)

Polymorphism



Gk. πολύμορφή
poly (many),
morph (shape)

- Numeric Coercions
- Subclass Polymorphism
- Generics
  - parametric polymorphism
  - generic classes & methods

Part 1 – Coercions & Autoboxing

- Widening coercions
  - `int` → `float`
  - `float` → `double`
  - `1 + 2.0 =`
  - `1 / 2 =`
  - `1 / 2.0 =`

- Narrowing coercions
  - require casts
  - `int i = (int) 1/ 2.0;`

Auto-boxing

- Quick Recap: Objects versus Primitives
  - **`int`** v. **`Integer`**
  - **`int`** is a primitive
    - Passed by value
  - **`Integer`** is an object
    - Passed by reference

- **`List<int>`** ✗
- **`List<Integer>`** ✓

Auto-boxing

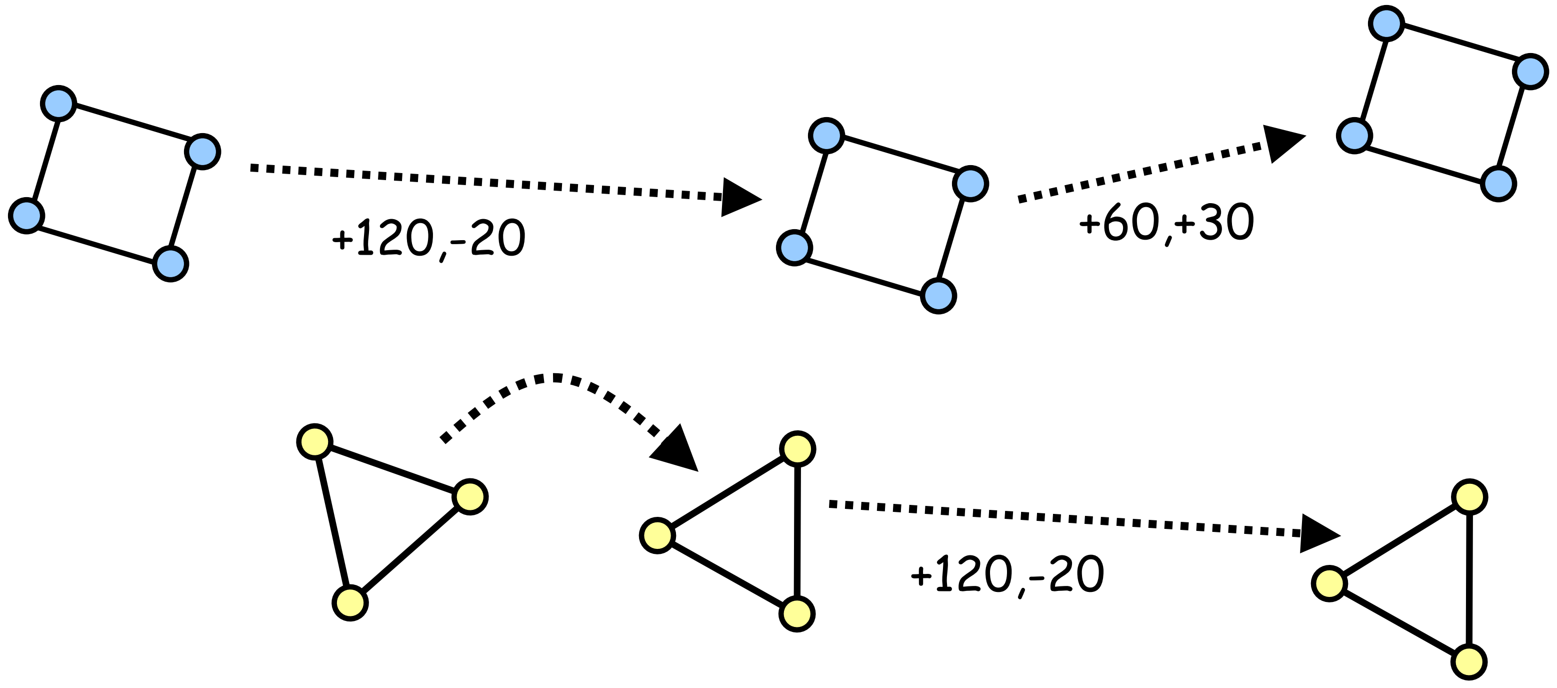
- Auto-boxing: the **automatic conversion** of **primitive types** to their corresponding **class types**

- `Integer i = 4;`

- `int i = new Integer(5);`

-
  ```
  List<Integer> list = …;
  list.add(75);
  int i = list.get(0);
  4.5 + list.get(0);
  ```

Part 2 – Subclass Polymorphism



- Treating different things in the same way!
  - e.g. a method for moving or rotating shapes shouldn't worry about what shape it is working with

Q) What's wrong with this?

```
class Weight {
    static int weightOfCat(Cat c) {
      if (c instanceof NinjaKitten)
        return 8;
      if (c instanceof Kitten)
        return 10;
      if (c instanceof Cat)
        return 20;
      return 0;
    }
}
```

Mental Model of Typing

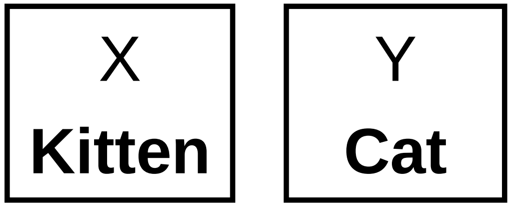
```
class Cat { … }
class Kitten extends Cat { … }

Kitten x = new Kitten();
Cat y = x;
```

Static (or Declared) type of y is "Cat"

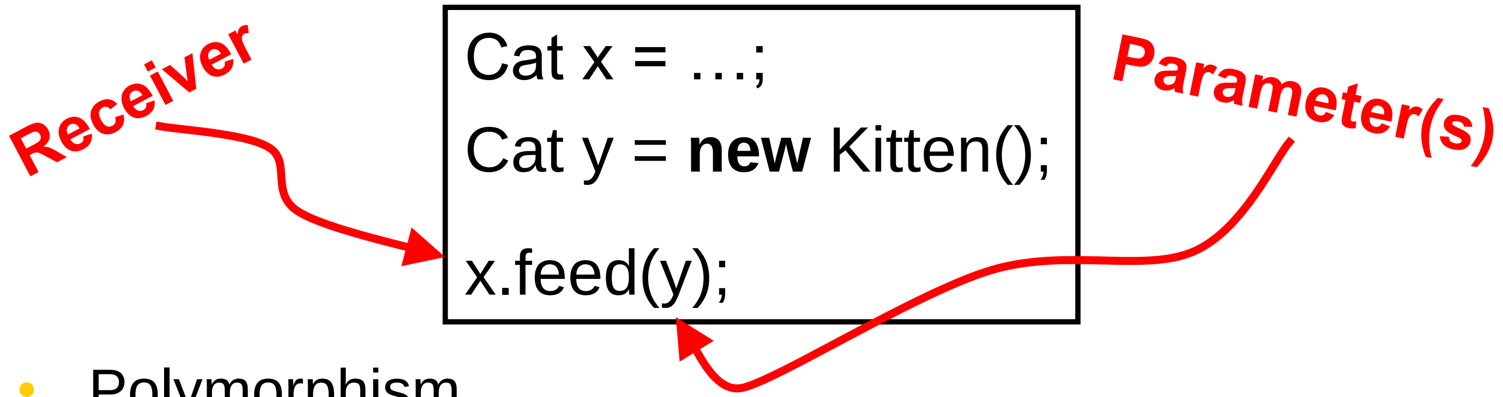
Dynamic (or Runtime) type of x and y is "Kitten"

- Static Type
  - **declared** type of a variable
- Dynamic Type
  - type of object **referenced** by variable



Dynamic Dispatch

- Dynamic dispatch
  - The **mechanism** which supports writing **generic code**



```
Cat x = …;
Cat y = new Kitten();

x.feed(y);
```

- Polymorphism
  - two aspects – compile time (static) and runtime (dynamic)
- Static checking phase
  - can only call methods defined in static type of receiver
  - based on static types of receiver and parameters
- Dynamic dispatch
  - selection of method at runtime
  - based on dynamic type of receiver (only)

Why dynamic dispatch?

```
class HouseCat {
 …              // lots of methods …
 void speak() { System.out.println("Meow!");
}}
class Tiger extends HouseCat {
 void speak() {
  System.out.println("ROOOOAAARRRR");
}}
```

- Dynamic dispatch + subclassing
  - Allows different object types to be used uniformly
  - Subclass behaviour should be compatible
    - E.g. a Tiger behaves like a HouseCat, except it's LOUDER!

Quiz: what gets printed?

```
class Cat {
 String whatAmI() {
  return “I’m a Cat!”;
}}

class Kitten extends Cat {
 String whatAmI() {
  return “I’m a Kitten!”;
}}

Cat gypsy = new Cat();
Cat spike = new Kitten();

System.out.println(“Gypsy: ” + gypsy.whatAmI());
System.out.println(“Spike: ” + spike.whatAmI());
```

A)
```
Gypsy: “I’m a Kitten!”
Spike: “I’m a Kitten!”
```

B)
```
Gypsy: “I’m a Cat!”
Spike: “I’m a Kitten!”
```

C)
```
Gypsy: “I’m a Cat!”
Spike: “I’m a Cat!”
```

More Dispatch Examples

```
class Cat {
 String whatAmI() {
  return “I’m a Cat!”;
}}

class Kitten extends Cat {
 String whatAmI() {
  return “I’m a Kitten!”;
}}

class NinjaKitten extends Kitten {
 String isKickedBy(Kitten k) { return “Ouch!”; }
}

Cat bob = new NinjaKitten();
System.out.println(“Bob: ” + bob.whatAmI());
```

A) `Bob: “I’m a Kitten!”`

B) `Bob: “Ouch!”`

C) **`error`**

More Dispatch Examples

```
class Cat {
 String whatAmI() {
  return "I'm a Cat!";
}}

class Kitten extends Cat {
 String whatAmI() {
  return "I'm a Kitten!";
}}

class NinjaKitten extends Kitten {
 String isKicked() { return "Ouch!"; }
}

Cat bob = new NinjaKitten();
System.out.println("Bob: " + bob.isKicked());
```

A) `Bob: "I'm a Kitten!"`

B) `Bob: "Ouch!"`

C) **`error`**

More Dispatch Examples

```
class Cat {
 String whatAmI() {
  return “I’m a Cat!”;
 }
 void print() {
  System.out.println(whatAmI());
}}
class Kitten extends Cat {
 String whatAmI() {
  return “I’m a Kitten!”;
}}
Cat gypsy = new Cat();
Cat spike = new Kitten();
gypsy.print();
spike.print();
```

A) “I’m a Kitten!”
   “I’m a kitten!”

B) “I’m a Cat!”
   “I’m a Kitten!”

C) “I’m a Cat!”
   “I’m a Cat!”

Quiz

```
class Cat {
 public void isClawedBy(Cat c) {
  System.out.println("Clawed by a Cat!");
}
 public void isClawedBy(Kitten c) {
  System.out.println("Clawed by a Kitten!");
}}
class Kitten extends Cat {}
Cat gypsy = new Cat();
Cat spike = new Kitten();
Kitten teddy = new Kitten();
gypsy.isClawedBy(spike);
spike.isClawedBy(teddy);
teddy.isClawedBy(teddy);
```

A)
```
"Clawed by a Cat!"
"Clawed by a Kitten!"
"Clawed by a Kitten!"
```

B)
```
"Clawed by a Cat!"
"Clawed by a Cat!"
"Clawed by a Kitten!"
```

Quiz

```
class Cat {
 public void isClawedBy(Cat c) {
  System.out.println("Clawed by a Cat!");
}}
class Kitten extends Cat {
 public void isClawedBy(Kitten k) {
  System.out.println("Clawed by a Kitten!");
}}

Cat gypsy = new Cat();
Cat spike = new Kitten();
Kitten teddy = new Kitten();
gypsy.isClawedBy(teddy);
spike.isClawedBy(teddy);
teddy.isClawedBy(teddy);
```

A)
```
"Clawed by a Cat!"
"Clawed by a Kitten!"
"Clawed by a Kitten!"
```

B)
```
"Clawed by a Cat!"
"Clawed by a Cat!"
"Clawed by a Kitten!"
```

Quiz

```
class Cat {
 public void isClawedBy(Cat c) {
  System.out.println("Clawed by a Cat!");
}}
class Kitten extends Cat {
 public void isClawedBy(Kitten k) {
  System.out.println("Clawed by a Kitten!");
}}

Cat gypsy = new Cat();
Kitten spike = new Kitten();
Kitten teddy = new Kitten();
gypsy.isClawedBy(teddy);
spike.isClawedBy(teddy);
teddy.isClawedBy(teddy);
```

A)
```
"Clawed by a Cat!"
"Clawed by a Kitten!"
"Clawed by a Kitten!"
```

B)
```
"Clawed by a Cat!"
"Clawed by a Cat!"
"Clawed by a Kitten!"
```

Summary

- Numeric Coercions & Autoboxing

- Subclass Polymorphism
  - enabled by inheritance
  - supported by typing rules and dynamic dispatch
  - facilitates generic code
  - key part of OO

Inheritance + Constructors

- Constructors are not inherited
- Constructors use super in first line to forward construction to super class
  - If the programmer does not explicitly write the super call, this call is added by the compiler

Implicit Constructor Code

```
class A { }
class B extends A {
  B(){
    System.out.println("B constructor");
}}
```

How your code looks like

```
class A extends Object {
  A() { super(); }
}


class B extends A {
  B(){
    super();
    System.out.println("B constructor");
}}
```

What is added implicitly