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

SWEN221:
Software Development

13: Encapsulation

David J. Pearce & Nicholas Cameron & James Noble & Petra Malik
Computer Science, Victoria University

Encapsulation

Encapsulate:

1. Enclose in a capsule or other small container.

Webster's online dictionary

- Objects have interfaces and implementations
  - Interface is external view of object
  - Implementation is "inner workings" of object
- An Object's implementation can change without affecting rest of system
  - Implementation must be invisible from outside

Understanding Encapsulation

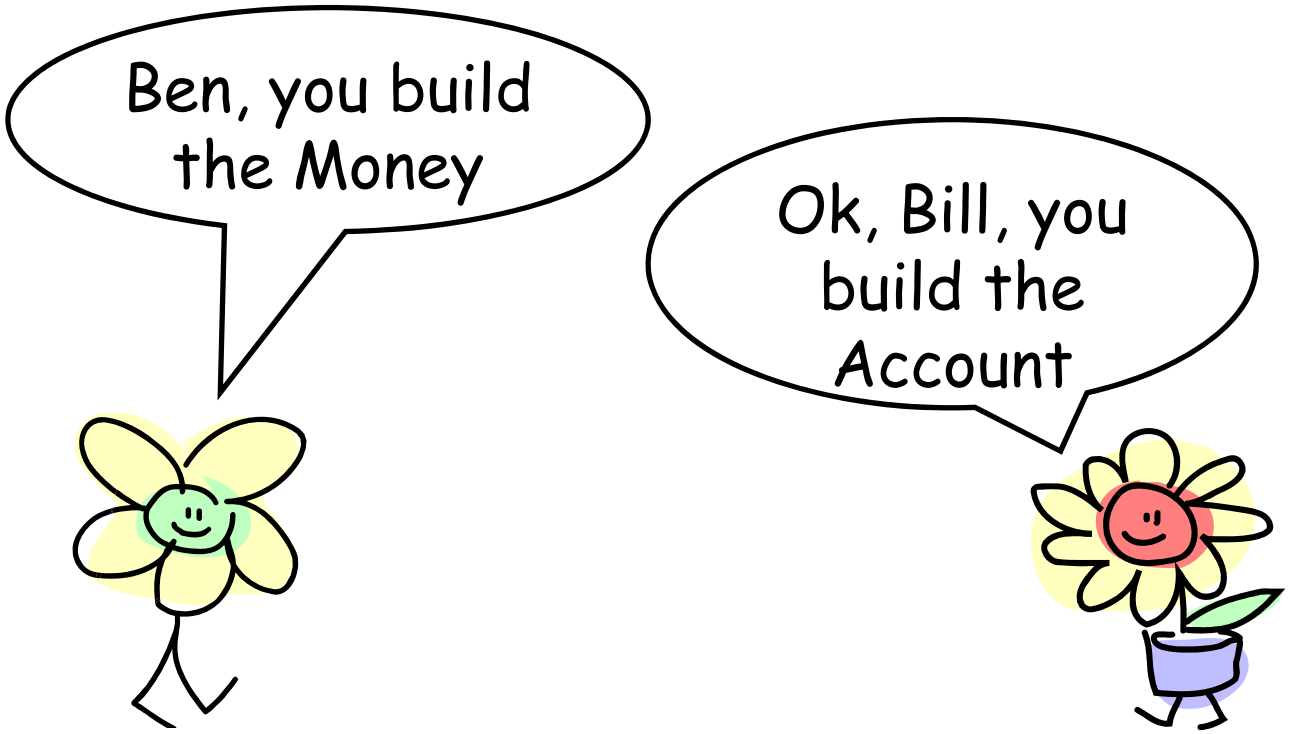
- Bike should be encapsulated:
  - Mechanic knows how it works
  - Rider does not

Understanding Encapsulation

- Encapsulating bike has other advantages:
  - Can change bike without affecting rider
  - Bike's might have different characteristics and implementations, but interface is the same

Understanding Encapsulation

- Bill & Ben build an accounting system …



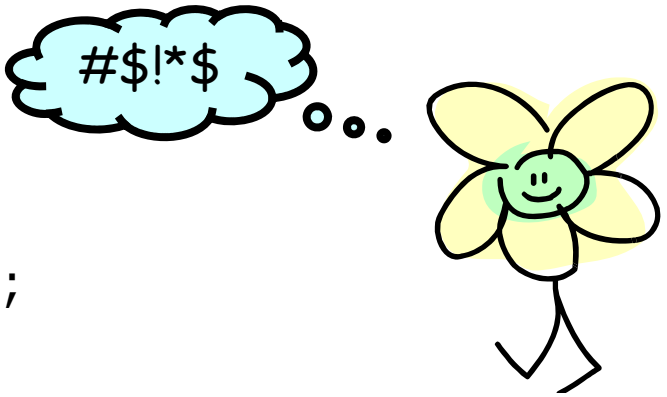
Understanding Encapsulation

```
class Money {
 public int dollars;
 public int cents; // cents < 100 must always hold
 …
}
```

Understanding Encapsulation


```
class Money {
 public int dollars;
 public int cents; // cents < 100 must always hold
 …
}
```

```
class Account {
 int balance; // in cents
 …
 void deposit(Money m) {
  balance += (m.dollars*100) + m.cents;
 }
 Money getBalance() {
  Money r = new Money();
  r.dollars = 0;
  r.cents = balance;
  return r;
}}
```



Breaks Money's invariant

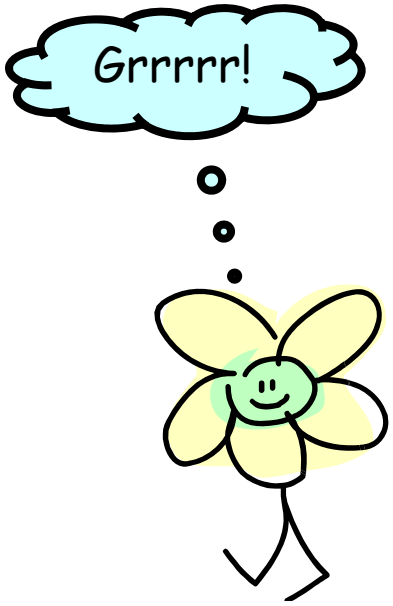
Understanding Encapsulation

- Meanwhile …

Understanding Encapsulation

```
class Money {
 public int dollars;
 public int cents; // cents < 100 must always hold
 …
}
```

```
class Account {
 int balance; // in cents
 …
 void deposit(Money m) {
  balance += (m.dollars*100) + m.cents;
 }
 Money getBalance() {
  Money r = new Money();
  r.dollars = 0;
  r.cents = balance;
  return r;
}}
```



Doesn't work now

Abstraction and Encapsulation

- **Abstraction**
  - An object simulates a part of the domain
  - Hiding — **abstracting** — the internal details
- **Encapsulation**
  - Objects can be changed only from the inside
  - Each change should keep the object consistent
  - That is, object *invariant's* must be maintained
- **How?**
  - Encapsulation boundary around objects' implementations
  - **public** — can be accessed anywhere
  - **private** — only from the same class
  - **package** — only from the same package
  - **protected** — class and subclasses (and package)

Maintaining Object Consistency

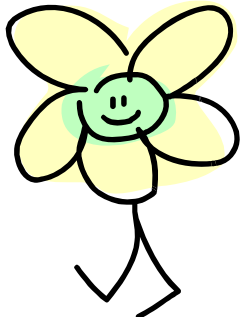
- Interface:
  - **Public** messages (i.e. methods)
  - Hides true receiver (i.e. receiver's class)
  - Accessors – do not change object state
  - Mutators – may change object state
- Implementation:
  - **Private** (or **protected**) methods and fields
  - Beware of **protected** fields
- If users need access to fields:
  - Provide public getter / setter methods
  - Do not make fields **public**

Sorting it out

```
class Money {
 private int dollars;
 private int cents; // cents < 100 must always hold

 public Money(int d, int c) {
  if(c>99 || c<0) throw IllegalArgumentException();
  dollars=d; cents=c;
 }
 public int getDollars() { return dollars; }
 public int getCents() { return cents; }
}
```

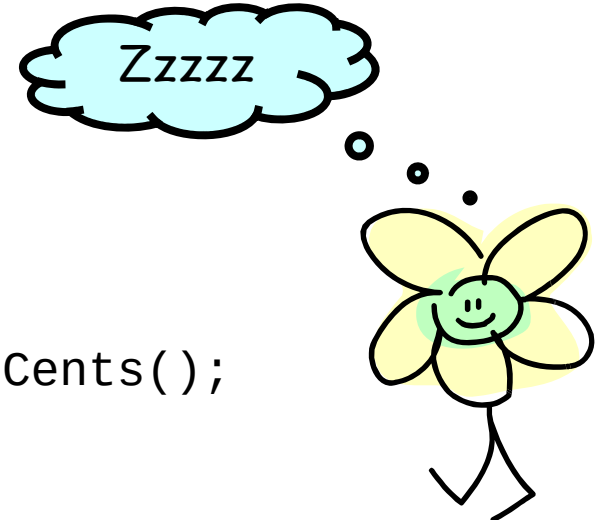
```
class Account {
 private int balance; // in cents
 …
 void deposit(Money m) {
  balance += (m.getDollars()*100) + m.getCents();
}}
```



Ben has a bright idea!

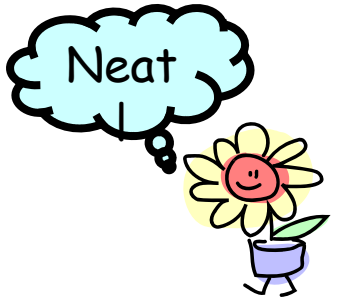
```
class Money {
 private int dollars;
 private int cents; // cents < 100 must always hold

 public Money(int d, int c) {
  if(c>99 || c<0) throw IllegalArgumentException();
  cents=c + (d*100);
 }
 public int getDollars() { return cents / 100; }
 public int getCents() { return cents % 100; }
}
class Account {
 private int balance; // in cents
 …
 void deposit(Money m) {
  balance += (m.getDollars()*100) + m.getCents();
}}
```

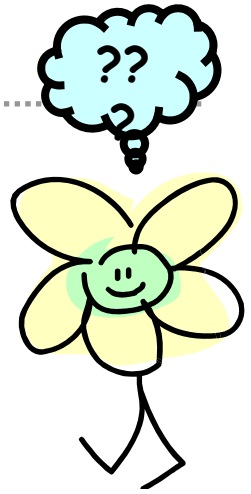


Beware of breaking encapsulation!

```
interface Money {
 int getCents();
 void setCents(int c); …  // and for dollars
}
class CentsOnly implements Money {
 private int cents; …
}
class DollarsAndCents implements Money {
 private int cents;   // cents < 100 always holds
 private int dollars; …
}
```

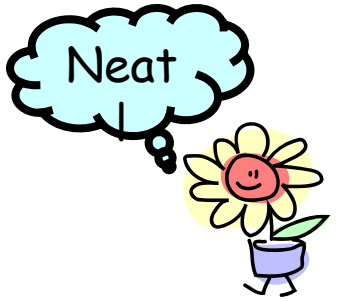



```
class Account {
 private CentsOnly balance;
 …
 public CentsOnly getBalance() { return balance; }
}
```



Beware of breaking encapsulation!

```
Interface Money {
 int getCents();
 void setCents(int c); …  // and for dollars
}
class CentsOnly implements Money {
 private int cents; …
}
class DollarsAndCents implements Money {
 private int cents;   // cents < 100 always holds
 private int dollars; …
}
```




```
class Account {
 private Money balance;
 …
 public Money getBalance() { return balance.clone(); }
}
```