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


# SWEN221 Software Development

# Encapsulation

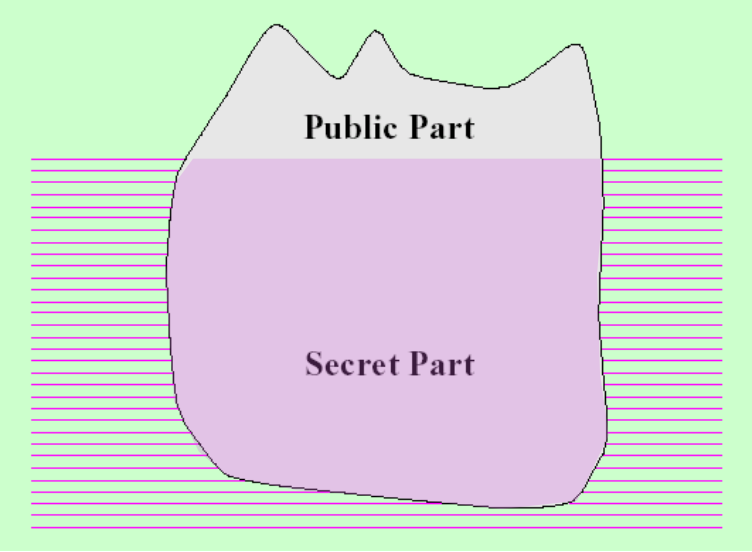
Thomas Kuehne

Victoria University

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

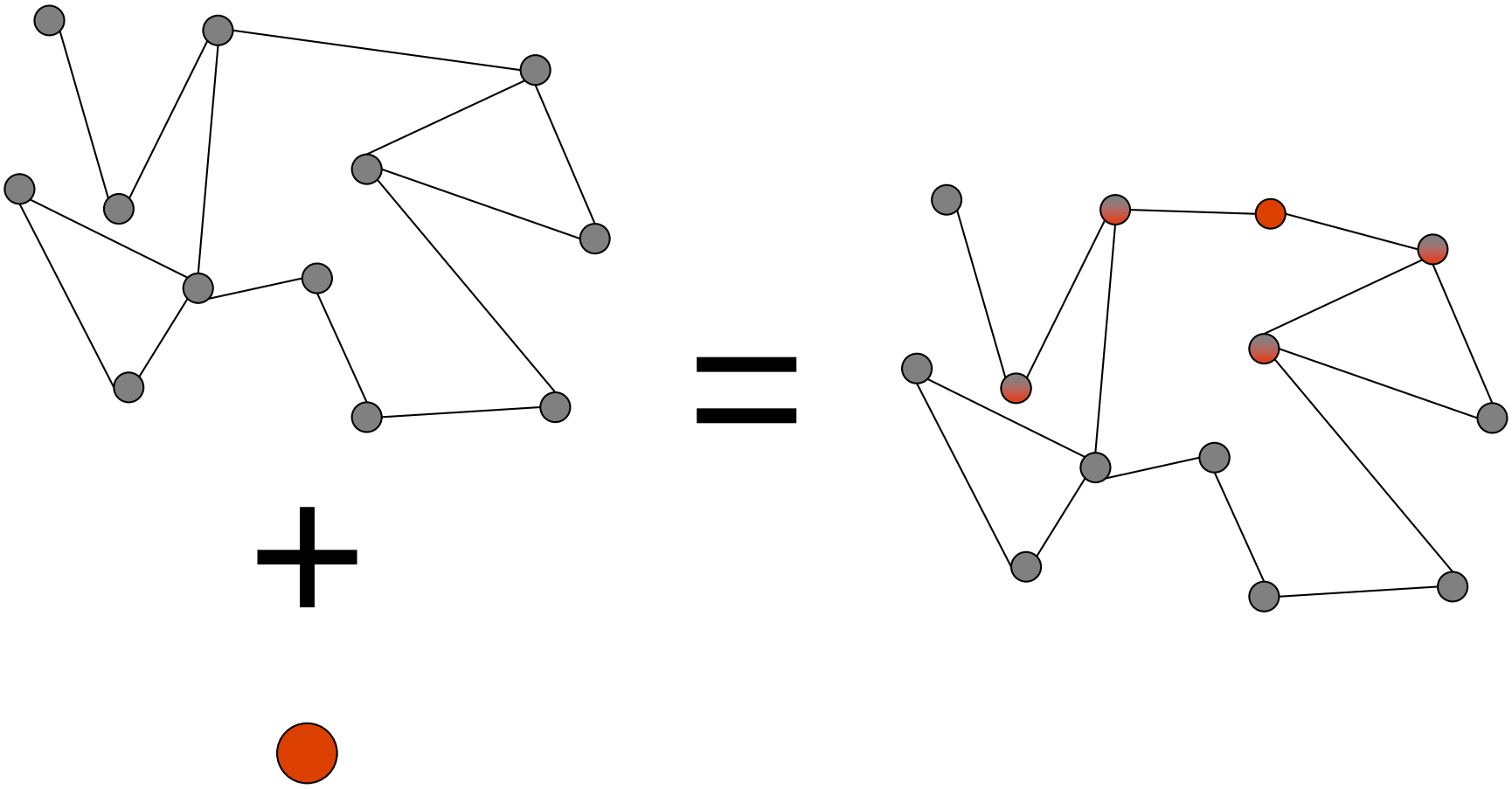
# Information Hiding

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules.

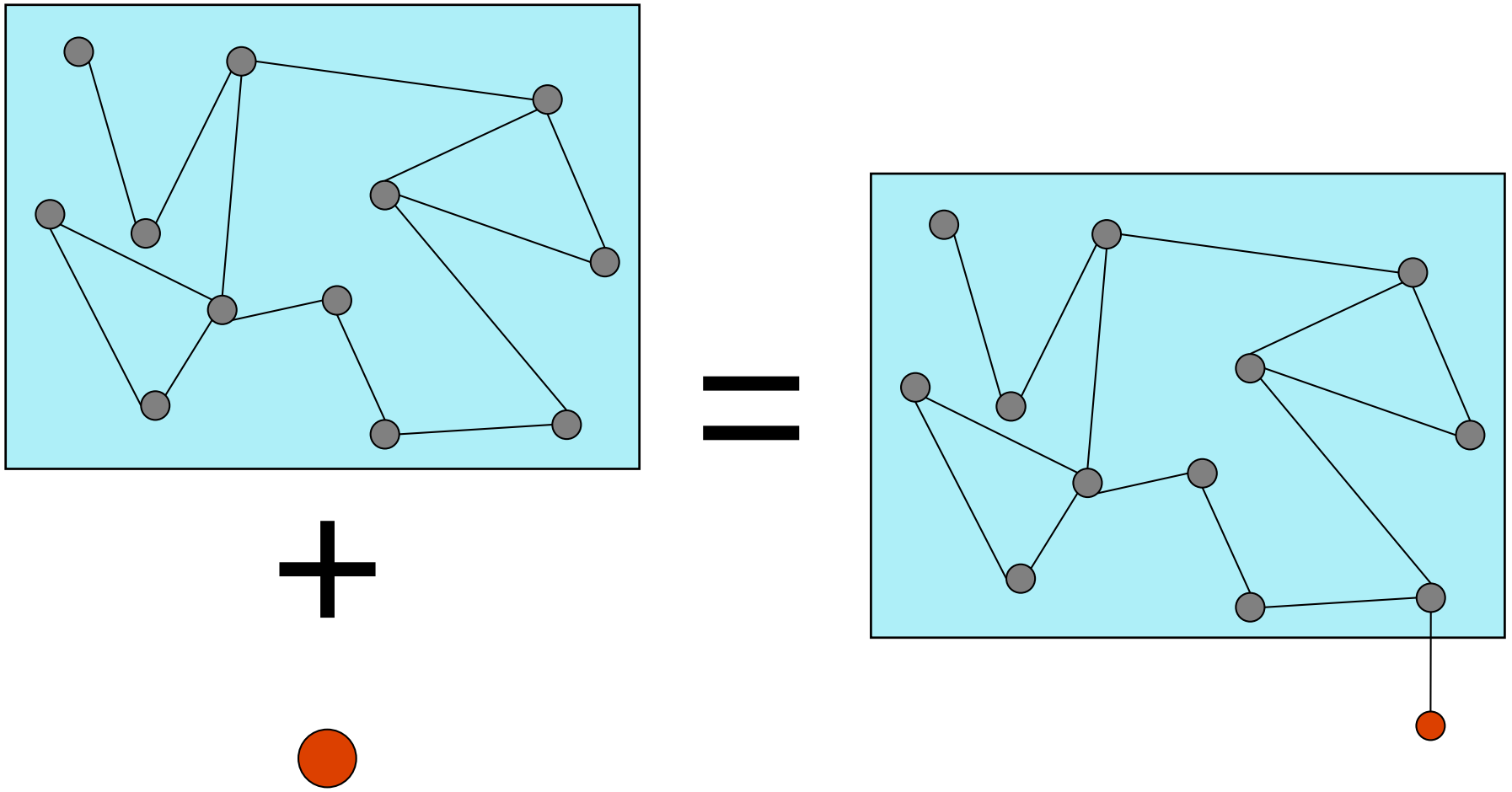



prevents ripple on effects caused by clients depending on internal details

# Bad Extensibility



# Good Extensibility

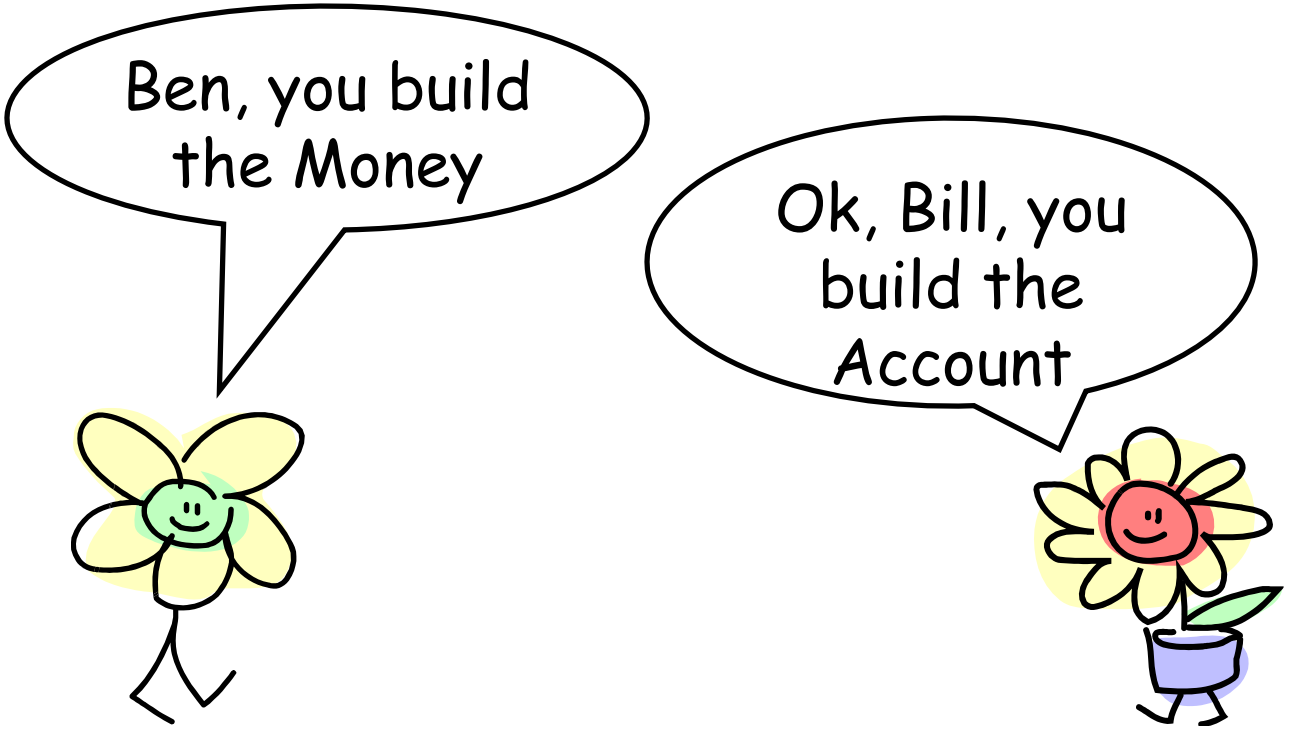


# Encapsulation

- A mechanism for information hiding
  - offered by typed and untyped OO languages
- Controlling access to features
  - explicit getters/setters required
  - visibility constraints
  - fields and operations
- Ultimate Goal: Reducing Dependencies
  - local implementation choices should not affect clients

# Unfortunate Series of Events…

- Bill & Ben build an accounting system …



# Unfortunate Series of Events…

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

# Unfortunate Series of Events…

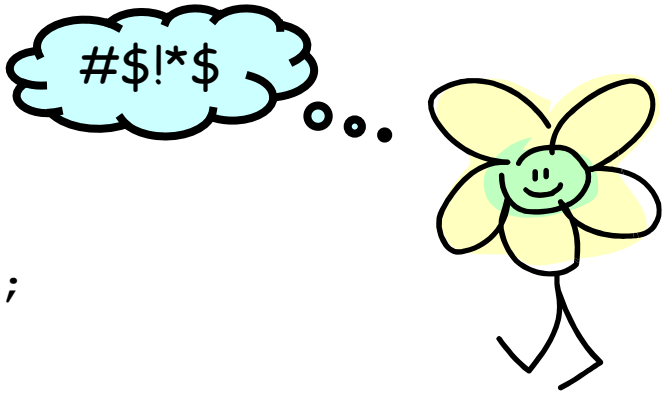
```
class Money {
 public int dollars;
 public int cents; // cents < 100 must always hold
 …
}
```

```
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;
}}
```

# Unfortunate Series of Events…


```
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

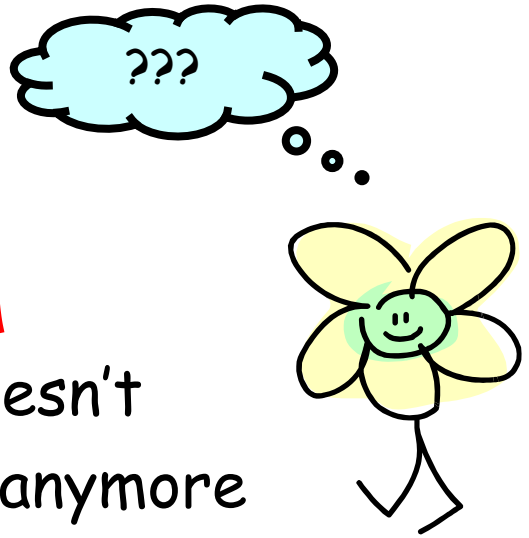
# Unfortunate Series of Events…

- Meanwhile …

# Unfortunate Series of Events

```
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 = balance / 100;
  r.cents = balance % 100;
  return r;
}}
```

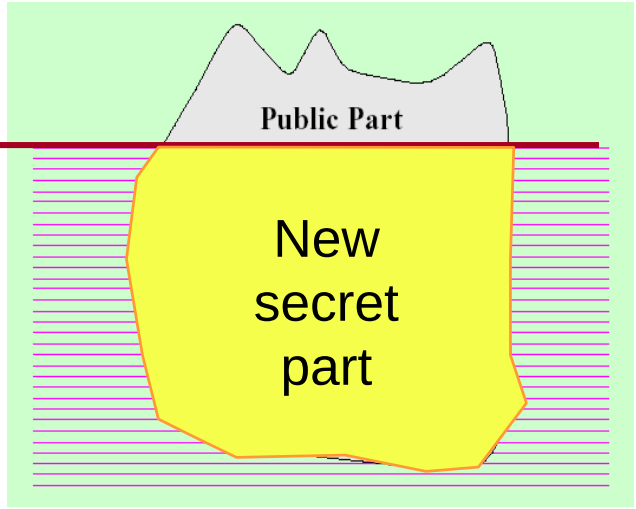



# Encapsulation Foundations

- **Information Hiding**
  - shielding clients from ripple on effects
- **Maintaining Invariants**
  - state can only be manipulated from the inside
  - each change should keep the object consistent
- **Typical language mechanism**
  - Visibility constraints
  - **public** — can be accessed anywhere
  - **private** — only from the same class
  - **package** — only from the same package
  - **protected** — class and subclasses (and package)

# Hiding Implementation Detail

```
class Queue<Element>
{
  public int size();
  public insert(Element e);
  ---------------------------
  private Element elements[];
  private putAt(int pos, Element e);
  ...
}
```



# Encapsulation Challenges

- Is hiding instance variables (and methods) sufficient?
  - what about getters and setters?
  - what about leaking state?
  - what about maintaining invariants?
  - what about leaking internal types?
  - what about code organization?

# Leaking Types and State

- Java's **"private"** keyword doesn't guarantee encapsulation
  - Can "leak" references to internal state

```
class Shape {
 private ArrayList<Point> points = …;
 …
 public ArrayList<Point> getPoints() {
   return points;
 }
}
```

# Maintaining Invariants

## Getters & Setters

```
double amount;

amount = account.balance()
amount -= 200;
account.setBalance(amount);
```

**potential of exposing internal representation**

**account possibly manipulated by others in between (not only in a concurrent setting)**

## Consistent Operations

```
account.withdraw(200);
```

# Unnecessarily Exposing Secrets

## Sending a Sequence of Messages

```
Account a;

a = bank.customer("fred").account();
```

## Law of Demeter

```
a = bank.accountFor("fred");
```

**not the business of the client to know that banks have "customers"**

---

**return type of "customer()" could change**

# Messages as Goals

## Getters & Setters

```
String d="11/04/2017";

book.loanDate(d);
book.checkedOut(true);
book.borrowedBy("Joe");
```

**unnecessary dependency on date representation**

---

**potential of missing operations**

## Messages as Goals

```
book.checkOutFor("Joe");
```

**method will create a date itself, ensuring accuracy of information!**

# Where Should Code Go?

## Violating the Dilbert Principle

```
int total = 0;
for (Billing b : plant.billings()) {
  if (b.status() == Billing.PAID && b.date > startDate)
    total += b.amount();
}
```

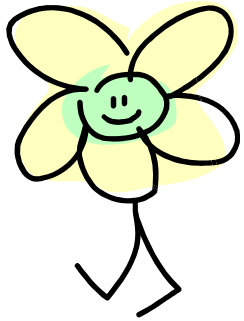
## Let someone else do the work for you

```
total = plant.totalBillingsPaidSince(startDate);
```

# Example Revisited

```
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; }
}
```

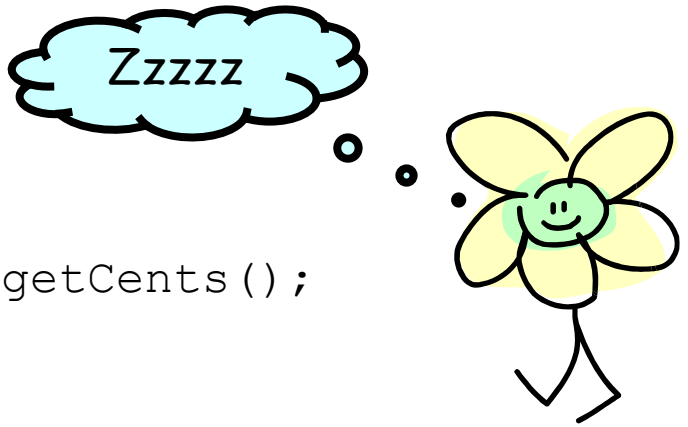
```
class Account {
 private int balance; // in cents
 …
 void deposit(Money m) {
  balance += (m.getDollars()*100) + m.getCents();
}}
```



# Ben has a bright idea!

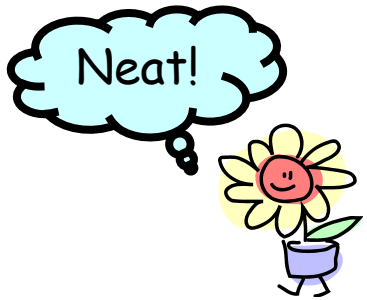
```
class Money {
 private int dollars;
 private int cents; // cents < 100 must always hold

 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();
}}
```



# Further Improving Maintainability

```
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(); }
}
```