Chain Constructors *

You have multiple constructors
that contain duplicate code

Chain the constructors together
to obtain the least duplicate code

```
public class Loan {
  ...
  public Loan(float notional, float outstanding, int rating, Date expiry) {
    this.strategy = new TermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating =rating;
    this.expiry = expiry;
  }
  public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this.strategy = new RevolvingTermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
  }
  public Loan(CapitalStrategy strategy, float notional, float outstanding,
                  int rating, Date expiry, Date maturity) {
    this.strategy = strategy;
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
  }
}
```

```
public class Loan {
  ...
  public Loan(float notional, float outstanding, int rating, Date expiry) {
    this(new TermROC(), notional, outstanding, rating, expiry, null);
  }
  public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this(new RevolvingTermROC(), notional, outstanding, rating, expiry, maturity);
  }
  public Loan(CapitalStrategy strategy, float notional, float outstanding,
                  int rating, Date expiry, Date maturity) {
    this.strategy = strategy;
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
  }
}
```

Motivation

Code that's duplicated across two or more of a class's constructors is an invitation for trouble. Someone adds a new variable to a class, updates a constructor to initialize the variable, but neglects to update the other constructors, and bang, say hello to your next bug. The more constructors you have in a class, the more duplication will hurt you. It's therefore a good idea to reduce or remove all duplication if possible, which has the added bonus of reducing your system's *code bloat*.

We often accomplish this refactoring with *constructor chaining*: specific constructors call more general-purpose constructors until a final constructor is reached. If you have one constructor at the end of every chain, I call that your *catch-all* constructor, since it handles every constructor call. This catch-all constructor often accepts more parameters than the other constructors, and may or may not be private or protected.

If you find that having many constructors on your class detracts from its usability, consider *Replace Multiple Constructors with Factory Methods*.

Communication	Duplication	Simplicity
When constructors in a class implement duplicate work, the code fails to communicate what is specific from what is general. Communicate this by having specific constructors forward calls to more general-purpose constructors and do unique work in each constructor.	Duplicate code in your constructors makes your classes more error-prone and harder to maintain. Find what is common, place it in general-purpose constructors, forward calls to these general constructors and implement what isn't general in each constructor.	If more than one constructor contains the same code, it's harder to see how each constructor is different. Simplify your constructors by making specific ones call more general purpose ones, in a chain.

Mechanics

1. Find two constructors (called A and B) that contain duplicate code. Determine if A can call B or if B can call A, such that the duplicate code can be safely (and hopefully easily) deleted from one of the two constructors.

2. Compile and test.

3. Repeat steps 1 and 2 for all constructors in the class, including ones you've already touched, in order to obtain as little duplication across all constructors as possible.

4. Change the visibility of any constructors that may not need to be public.

5. Compile and test.

Example

1. We'll go with the example shown in the code sketch. We start with a single Loan class, which has three constructors to represent different types of loans and tons of bloated and ugly duplication:

```
public Loan(float notional, float outstanding, int rating, Date expiry) {
    this.strategy = new TermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
}
```

```
public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this.strategy = new RevolvingTermROC();
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
}

public Loan(CapitalStrategy strategy, float notional, float outstanding, int rating,
            Date expiry, Date maturity) {
    this.strategy = strategy;
    this.notional = notional;
    this.outstanding = outstanding;
    this.rating = rating;
    this.expiry = expiry;
    this.maturity = maturity;
}
```

I study the first two constructors. They do contain duplicate code, but so does that third constructor. I consider which constructor it would be easier for the first constructor to call. I see that it could call the third constructor, with a minimum about of work. So I change the first constructor to be:

```
public Loan(float notional, float outstanding, int rating, Date expiry) {
    this(new TermROC(), notional, outstanding, rating, expiry, null);
}
```

2. I compile and test to see that the change works.

3. I repeat steps 1 and 2, to remove as much duplication as possible. This leads me to the second constructor. It appears that it too can call the third constructor, as follows:

```
public Loan(float notional, float outstanding, int rating, Date expiry, Date maturity) {
    this(new RevolvingTermROC(), notional, outstanding, rating, expiry, maturity);
}
```

I'm now aware that constructor three is my class's catch-all constructor, since it handles all of the construction details.

4. I check all callers of the three constructors to determine if I can change the public visibility of any of them. In this case, I can't (take my word for it – you can't see the code that calls these methods).

5. I compile and test to complete the refactoring.