Imperative Programming 2:
Abstract members

Hongseok Yang
University of Oxford

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

- When would you use A1? What about A2?

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

- When would you use A1? What about A2?
- Who provides the less parameters of A1 and A2?

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

- When would you use A1? What about A2?
- Who provides the less parameters of A1 and A2?

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

Client program of A1

```
val a = Array(3,2,1)

val s1 = new A1; s1 sort (a, _<_); println(a mkString ",")
```

- When would you use A1? What about A2?
- Who provides the less parameters of A1 and A2?

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

Client program of A1

```
val a = Array(3,2,1)

val s1 = new A1; s1 sort (a, _<_); println(a mkString ",")

object s2 extends A2 { def less(x1:Int, x2:Int) = x1 > x2 }
s2 sort a; println(a mkString ",")
```

Subclass of A2

Two notions of abstraction

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods

Two notions of abstraction

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  def less(x1: Int, x2: Int): Boolean
  def sort(a: Array[Int]) = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods

Two notions of abstraction

```
class A1 {
  def sort(a: Array[Int], less: (Int,Int) => Boolean) = { ... }
}

abstract class A2 {
  val a: Array[Int]
  def less(x1: Int, x2: Int): Boolean
  def sort() = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods, abstract values

Two notions of abstraction

```
class A1[T] {
  def sort(a: Array[T], less: (T,T) => Boolean) = { ... }
}

abstract class A2 {
  val a: Array[Int]
  def less(x1: Int, x2: Int): Boolean
  def sort() = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods, abstract values

Two notions of abstraction

```
class A1[T] {
  def sort(a: Array[T], less: (T,T) => Boolean) = { ... }
}

abstract class A2 {
  type T
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods, abstract values, abstract types

Two notions of abstraction

```
class A1[T <: List[Any]] {
  def sort(a: Array[T], less: (T,T) => Boolean) = { ... }
}

abstract class A2 {
  type T
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods, abstract values, abstract types

Two notions of abstraction

```
class A1[T <: List[Any]] {
  def sort(a: Array[T], less: (T,T) => Boolean) = { ... }
}

abstract class A2 {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

1. Parameterisation (i.e., parameter passing).
2. Abstract members -- abstract methods, abstract values, abstract types with bounds.

Usually, a client program of `A1` knows what `T`, `a` and `less` are, but a subclass of `A1` doesn't.

```
class A1[T <: List[Any]] {
  def sort(a: Array[T], less: (T,T) => Boolean) = { ... }
}

abstract class A2 {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

Usually, a subclass of `A1` knows what `T`, `a` and `less` are, but a client program of `A1` doesn't.

Learning outcome

- Can use abstract members in writing Scala programs.
- Can describe the meaning of abstract types.
- Can understand how to use abstract types for hiding implementation details.

Syntax: Declaration

- Just declare, but don't give a definition.
- It works for all of abstract values, methods and types.

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

Syntax: Instantiation

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

- Use overriding.

1. By subclassing or subtraiting.
2. By anonymous class.

Syntax: Instantiation

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

- Use overriding.

1. By subclassing or subtraiting.

2. By anonymous class.

```
class MySort extends Sort {
  type T = List[Int]
  val a:Array[T] =
    Array.fill(100)(Nil)
  def less(x:T, y:T) =
    x.length < y.length
}
val s: Sort = new MySort
```

Syntax: Instantiation

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() = { ... }
}
```

- Use overriding.

1. By subclassing or subtraiting.

2. By anonymous class.

```
class MySort extends Sort {
  type T = List[Int]
  val a:Array[T] =
    Array.fill(100)(Nil)
  def less(x:T, y:T) =
    x.length < y.length
}
val s: Sort = new MySort
```

```
val s: Sort =
  new Sort {
    type T = List[Int]
    val a: Array[T] =
      Array.fill(100)(Nil)
    def less(x: T, y:T) =
      x.length < y.length
  }
```

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() { a.sortWith(less _) }
  def compare(b: Array[T]): Boolean =
    (a zip b) forall (x => less(x._1, x._2))
}
class MySort extends Sort {
  type T = List[Int]
  val a: Array[T] = Array.fill(100)(Nil)
  def less(x:T, y:T) = x.length < y.length
}

val s1: MySort = new MySort; s1.sort()
val s2: MySort = new MySort; s2.sort()
println(s1 compare s2.a)
```

[Q1] What is the output?

1) `true` 2) `false` 3) random 4) compile error

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() { a.sortWith(less) }
  def compare(b: Array[T]): Boolean =
    (a zip b) forall (x => less(x._1, x._2))
}
class MySort extends Sort {
  type T = List[Int]
  val a: Array[T] = Array.fill(100)(Nil)
  def less(x:T, y:T) = x.length < y.length
}

val s1: MySort = new MySort; s1.sort()
val s2: MySort = new MySort; s2.sort()
println(s1 compare s2.a)
```

The type `T` of an `MySort` object is `List[Int]`, and this is known everywhere.

[Q1] What is the output?

1) `true` 2) `false` 3) random 4) compile error

```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() { a.sortWith(less) }
  def compare(b: Array[T]): Boolean =
    (a zip b) forall (x => less(x._1, x._2))
}
class MySort extends Sort {
  type T = List[Int]
  val a: Array[T] = Array.fill(100)(Nil)
  def less(x:T, y:T) = x.length < y.length
}

val s1: Sort = new MySort; s1.sort()
val s2: Sort = new MySort; s2.sort()
println(s1 compare s2.a)
```

[Q2] What is the output?

1) `true` 2) `false` 3) random 4) compile error

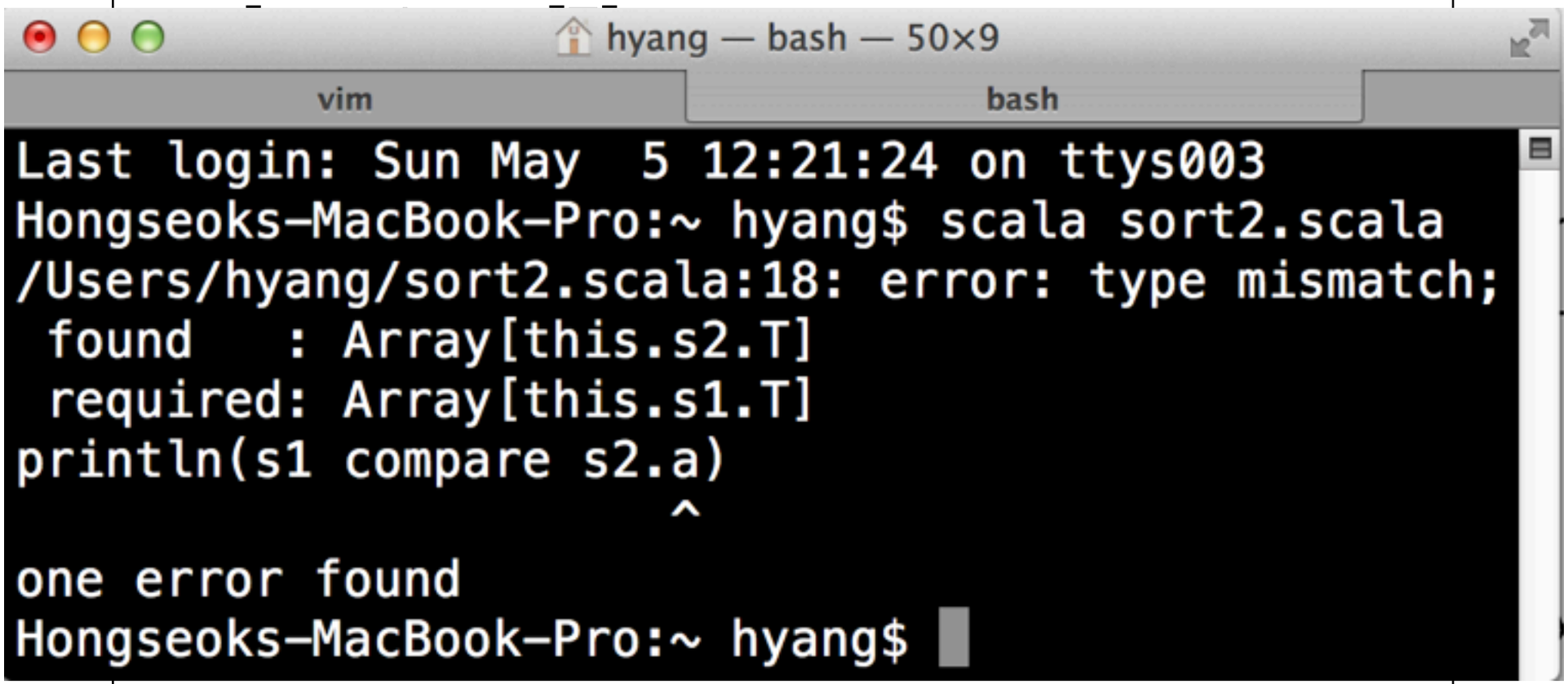
```
abstract class Sort {
  type T <: List[Any]
  val a: Array[T]
  def less(x1: T, x2: T): Boolean
  def sort() { a.sortWith(less) }
  def compare(b: Array[T]): Boolean =
    (a zip b) forall (x => less(x._1, x._2))
}
class MySort extends Sort {
  type T = List[Int]
  val a: Array[T] = Array.fill(100)(Nil)
  def less(x:T, y:T) = x.length < y.length
}

val s1: Sort = new MySort; s1.sort()
val s2: Sort = new MySort; s2.sort()
println(s1 compare s2.a)
```

Each object of type `Sort` has its own `T`, whose identity is unknown to others.

[Q2] What is the output?

1) `true` 2) `false` 3) random 4) compile error

```
abstract class Sort {
  type T <: List[Any]
```




```
val s1: Sort = new MySort; s1.sort()
val s2: Sort = new MySort; s2.sort()
println(s1 compare s2.a)
```

Each object of type `Sort` has its own `T`, whose identity is unknown to others.

[Q2] What is the output?

1) `true` 2) `false` 3) random 4) compile error

Semantics of abstract types

```
abstract class Sort {
  type T
  ...
  def compare(b: Array[T]): Boolean ...
}
class MySort extends Sort { ... }

val s1: Sort = new MySort
val s2: Sort = new MySort
// println(s1 compare s2.a) -- Compile Error
```

- New `T` for each object `s` of `Sort`.
- Thus, `s1.T` ≠ `s2.T`
- This feature is used for fine-tuning a spec about where a method can be used.

Path-dependent type "s.T"

```
abstract class Sort {
  type T
  ...
}
val s1: Sort = ...
val s2: Sort = ...
```

```
class Circuit {
  class Wire { ... }
  ...
}
val c1 = new Circuit
val c2 = new Circuit
var w1 = new c1.Wire
var w2 = new c2.Wire
```

- Type-per-object is also used for inner class.
- All such per-object types are denoted by path-dependent type expressions, such as `x.y.z.T`
- They are called path-dependent types.
- Examples -- `s1.T`, `s2.T`, `c1.Wire`, `c2.Wire`

Path-dependent type "s.T"

```
abstract class Sort {
  type T
  ...
}
val s1: Sort = ...
val s2: Sort = ...
```

```
class Circuit {
  class Wire { ... }
  ...
}
val c1 = new Circuit
val c2 = new Circuit
var w1 = new c1.Wire
var w2 = new c2.Wire
w1 = w2
```

compile error

- Type-per-object is also used for inner class.
- All such per-object types are denoted by path-dependent type expressions, such as `x.y.z.T`
- They are called path-dependent types.
- Examples -- `s1.T`, `s2.T`, `c1.Wire`, `c2.Wire`

Programming challenge: Currencies

- Build a library for currencies.
- At least US Dollar and UK Pound.
- Hide implementation details.
- Support money addition.
- No addition between dollars and pounds.

1. USD & GBP
2. Hiding
3. Addition
4. Don't mix

1. USD & GBP
~~2. Hiding~~
3. Addition
4. Don't mix

```
abstract class Currency ...

object Currency {
  private class UK ... extends Currency ...
  private class US ... extends Currency ...

  def makeUS(...): Currency = new US(...)
  def makeUK(...): Currency = new UK(...)
}
```

1. USD & GBP
2. ~~Hiding~~
3. Addition
4. Don't mix

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency ...
  override def toString = (amount +“ “+ designation)
}

object Currency {
  private class UK ... extends Currency ...
  private class US ... extends Currency ...

  def makeUS(...): Currency = new US(...)
  def makeUK(...): Currency = new UK(...)
}
```

1. USD & GBP
2. ~~Hiding~~
3. Addition
4. Don't mix

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency ...
  override def toString = (amount +“ “+ designation)
}

object Currency {
  private class UK(val amount: Long) extends Currency {
    val designation = “GBP“
    ...
  }
  private class US ... extends Currency ...

  def makeUS(...): Currency = new US(...)
  def makeUK(...): Currency = new UK(...)
}
```

~~1. USD & GBP~~
~~2. Hiding~~
3. Addition
4. Don't mix

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency ...
  override def toString = (amount +“ “+ designation)
}

object Currency {
  private class UK(val amount: Long) extends Currency {
    val designation = “GBP“
    ...
  }
  private class US(d: Long, p: Long) extends Currency {
    val amount = d * 100 + p
    val designation = “USD“
    ...
  }
  def makeUS(...): Currency = new US(...)
  def makeUK(...): Currency = new UK(...)
}
```

~~1. USD & GBP~~
~~2. Hiding~~
3. Addition
4. Don't mix

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency ...
  override def toString = (amount +“ “+ designation)
}

object Currency {
  private class UK(val amount: Long) extends Currency {
    val designation = “GBP“
    ...
  }
  private class US(d: Long, p: Long) extends Currency {
    val amount = d * 100 + p
    val designation = “USD“
    ...
  }
  def makeUS(a: Long): Currency = new US(a / 100, a % 100)
  def makeUK(a: Long): Currency = new UK(a)
}
```

[Q] Can you implement add?

~~1. USD & GBP~~

~~2. Hiding~~

3. Addition

4. Don't mix

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency ...
  override def toString = (amount +" "+ designation)
}

object Currency {
  private class UK(val amount: Long) extends Currency {
    val designation = "GBP"
    ...
  }
  private class US(d: Long, p: Long) extends Currency {
    val amount = d * 100 + p
    val designation = "USD"
    ...
  }
  def makeUS(a: Long): Currency = new US(a / 100, a % 100)
  def makeUK(a: Long): Currency = new UK(a)
}
```

[Q] Can you implement add?

~~1. USD & GBP~~
~~2. Hiding~~
~~3. Addition~~
4. Don't mix

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency
  override def toString = (amount +" "+ designation)
}

object Currency {
  private class UK(val amount: Long) extends Currer
    val designation = "GBP"
    def add(c: Currency) = makeUK(amount+c.amount)
  }
  private class US(d: Long, p: Long) extends Currency {
    val amount = d * 100 + p
    val designation = "USD"
    def add(c: Currency) = makeUS(amount+c.amount)
  }
  def makeUS(a: Long): Currency = new US(a / 100, a % 100)
  def makeUK(a: Long): Currency = new UK(a)
}
```

It mixes dollars and pounds

Key idea -- Abstract type

```
abstract class Currency {
  def amount: Long
  val designation: String
  def add(c: Currency): Currency
  override def toString = (amount +“ “+ designation)
}
```

Key idea -- Abstract type

```
abstract class CurrencyZone {
  type T <: Currency


  abstract class Currency {
    def amount: Long
    val designation: String
    def add(c: T): T
    override def toString = (amount +“ “+ designation)
  }
}
```

New spec -- add now accepts money in a particular yet unknown currency T.

Key idea -- Abstract type

Abstract factory method for T here, because T is unique per object.

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    def add(c: T): T
    override def toString = (amount +“ “+ designation)
  }
}
```

New spec -- add now accepts money in a particular yet unknown currency T.

Key idea -- Abstract type

Abstract factory method for T here, because T is unique per object.

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    def add(c: T): T = make(amount + c.amount)
    override def toString = (amount +“ “+ designation)
  }
}
```

New spec -- add now accepts money in a particular yet unknown currency T.

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}
```

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {




}
object US extends CurrencyZone {




}
```

Two T's -- one unique to UK, and another to the US obj. Hence, no mix in add.

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {
  class Pound(...) extends Currency ...


  type T = Pound

}
object US extends CurrencyZone {




}
```

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {
  class Pound(...) extends Currency ...


  type T = Pound
  def make(amount: Long) = new Pound(amount)
}
object US extends CurrencyZone {




}
```

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {
  class Pound(val amount:Long) extends Currency {
    val designation = “GBP“
  }
  type T = Pound
  def make(amount: Long) = new Pound(amount)
}
object US extends CurrencyZone {




}
```

[Q1] Complete the definition of the US object.

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {
  class Pound(val amount:Long) extends Currency {
    val designation = “GBP“
  }
  type T = Pound
  def make(amount: Long) = new Pound(amount)
}
object US extends CurrencyZone {




  type T = ...
  def make(amount: Long) = ...
}
```

[Q1] Complete the definition of the US object.

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {
  class Pound(val amount:Long) extends Currency {
    val designation = “GBP“
  }
  type T = Pound
  def make(amount: Long) = new Pound(amount)
}
object US extends CurrencyZone {
  class Dollar(d: Long, p: Long) extends Currency {
    val amount = 100 * d + p
    val designation = “USD“
  }
  type T = Dollar
  def make(amount: Long) = new Dollar(amount / 100, amount % 100)
}
```

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

object UK extends CurrencyZone {
  class Pound(val amount:Long) extends Currency {
    val designation = “GBP“
  }
  type T = Pound
  def make(amount: Long) = new Pound(amount)
}
object US extends CurrencyZone {
  class Dollar(d: Long, p: Long) extends Currency {
    val amount = 100 * d + p
    val designation = “USD“
  }
  type T = Dollar
  def make(amount: Long) = new Dollar(amount / 100, amount % 100)
}
```

[Q1] Complete the definition of the US object.
[Q2] Why no private?

```
abstract class CurrencyZone {
  type T <: Currency
  def make(amount: Long): T

  abstract class Currency {
    def amount: Long
    val designation: String
    ... }
}

private object UK extends CurrencyZone {
  class Pound(val amount:Long) extends Currency {
    val designation = “GBP“
  }
  type T = Pound
  def make(amount: Long) = new Pound(amount)
}
private object US extends CurrencyZone {
  class Dollar(d: Long, p: Long) extends Currency {
    val amount = 100 * d + p
    val designation = “USD“
  }
  type T = Dollar
  def make(amount: Long) = new Dollar(amount / 100, amount % 100)
}
```

[Q1] Complete the definition of the US object.
[Q2] Why no private?

```
val UKZone : CurrencyZone = UK
val USZone : CurrencyZone = US
```

Abstract type

- Can be used to hide implementation details.
- It is unique per each object.
- This unique-per-object semantics enables us to achieve a stronger guarantee using types.

Summary

- Abstraction in Scala can be achieved by parameter passing and abstract members.
- Abstract types have unique-per-object semantics.
- This semantics can be used to achieve a stronger guarantee about programs.
- Read Chapter 20.

Optional reading

Section 5 of the Scala design paper:

http://www.scala-lang.org/docu/files/ScalaOverview.pdf

```
abstract class SubjectObserver {
  type S <: Subject
  type O <: Observer
  abstract class Subject { this : S =>
    private var observers: List[O] = List()
    def subscribe(obs: O) =
      observers = obs :: observers
    def publish =
      for (val obs <- observers) obs.notify(this)
  }
  trait Observer {
    def notify(sub: S): unit
  }
}
```

```
object SensorReader extends SubjectObserver {
  type S = Sensor
  type O = Display
  abstract class Sensor extends Subject {
    val label: String
    var value: double = 0.0
    def changeValue(v: double) = {
      value = v
      publish
    }
  }
  class Display extends Observer {
    def println(s: String) = ...
    def notify(sub: Sensor) =
      println(sub.label + "␣has␣value␣" + sub.value)
  }
}
```