Object-Oriented Programming

5. More Composition and Inheritance

Filip Křikava

https://courses.fit.cvut.cz/BI-OOP/

Questions?

Warmup?

How can we make the following compile?

```
class Animal:
  def play = println(s"$this is playing")

class Dog extends Animal
class Cat extends Animal

class Zoo[A <: Animal](val animals: List[A]):
  def play: Unit = animals.foreach(_.play)
  def add[?](animal: ?): Zoo[?] = new Zoo[?](animal :: animals)


val dogs = new Zoo(List(new Dog))
val all = dogs.add(new Cat)
all.play
```

https://scastie.scala-lang.org/VNAnHoVASFOOEkg42cSDEg

How can we make the following compile?

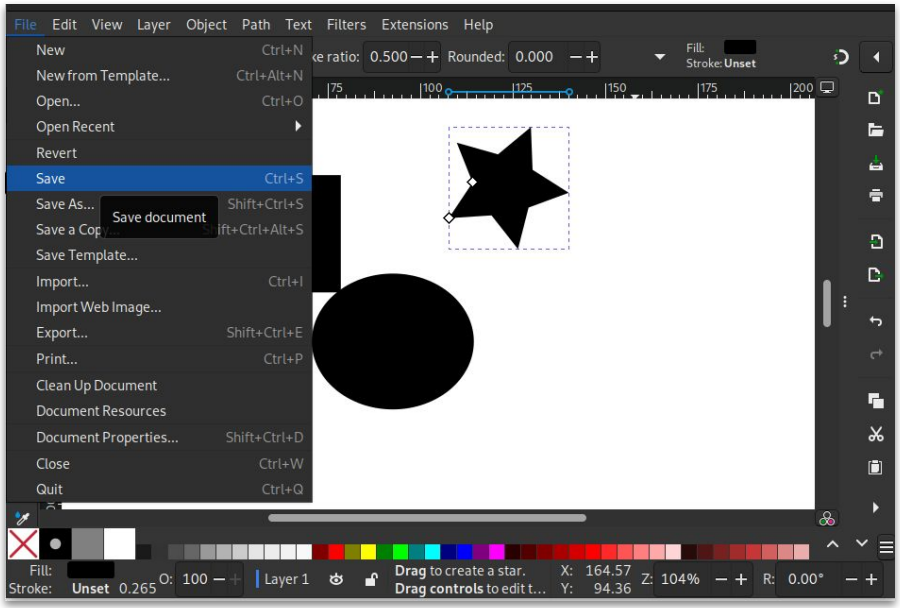
```
class Animal:
  def play = println(s"$this is playing")

class Dog extends Animal
class Cat extends Animal

class Zoo[A <: Animal](val animals: List[A]):
  def play: Unit = animals.foreach(_.play)
  def add[B >: A <: Animal](animal: B): Zoo[B] = new Zoo[B](animal :: animals)


val dogs = new Zoo(List(new Dog))
val all = dogs.add(new Cat)
all.play
```

https://scastie.scala-lang.org/FBaKJkzTTLW157Jt1IoOLw

Composition

Motivating example

- Remember the canvas with Shapes example?
- Let's add the ability to save the canvas to disk

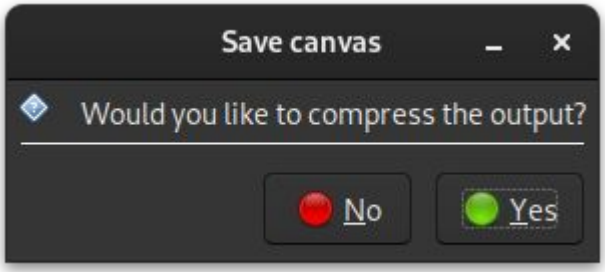


Motivating example

```
abstract class Shape

class Rectangle(w: Int, h: Int) extends Shape

class Canvas:
  val shapes = mutable.ArrayBuffer[Shape]()

class CanvasSerializer:
  def serialize(canvas: Canvas): Array[Byte] =
    for s <- canvas.shapes do
      s match
        case Rectangle(w, h) => …

class SaveAction(serializer: CanvasSerializer):
  def run(canvas: Canvas, output: File) =
    val bytes = serializer.serialize(canvas)
    Files.write(output.toPath, bytes)

val canvas = …
val serializer = CanvasSerializer()
val saveAction = SaveAction(serializer)
saveAction.run(canvas, File("/tmp/test"))
```

Motivating example

- Add support for output compression

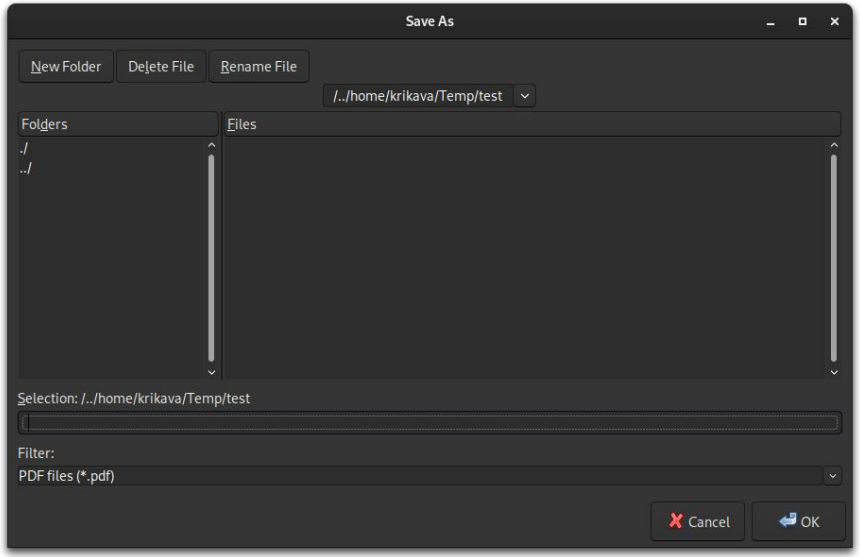


Motivating example

```
class CompressedCanvasSerializer(level: Int) extends CanvasSerializer:
  private def compress(bytes: Array[Byte]): Array[Byte] = ???

  override def serialize(canvas: Canvas): Array[Byte] =
    val bytes = super.serialize(canvas)
    compress(bytes)

class SaveAction(serializer: CanvasSerializer):
  def run(canvas: Canvas, output: File) =
    val bytes = serializer.serialize(canvas)
    Files.write(output.toPath, bytes)

val serializer = CompressedCanvasSerializer(1)

// since the CompressedCanvasSerializer is a CanvasSerializer
// we can use it in the SaveAction
// the subclassing does both - implementation and interface inheritance
val saveAction = SaveAction(serializer)
saveAction.run(canvas, File("/tmp/test"))
```

Motivating example

- Add support for different formats
  - PDF
  - SVG
- But keep the compression possibility



Motivating example

```
class CanvasSerializer:
  def serialize(canvas: Canvas): Array[Byte] = ???

class SVGCanvasSerializer extends CanvasSerializer:
  override def serialize(canvas: Canvas): Array[Byte] = ???

class PDFCanvasSerializer extends CanvasSerializer:
  override def serialize(canvas: Canvas): Array[Byte] = ???
```

- But what to do with the compression?

```
class CompressedCanvasSerializer(level: Int) extends CanvasSerializer:
  private def compress(bytes: Array[Byte]): Array[Byte] = ???

  override def serialize(canvas: Canvas): Array[Byte] =
    val bytes = super.serialize(canvas)
    compress(bytes)
```

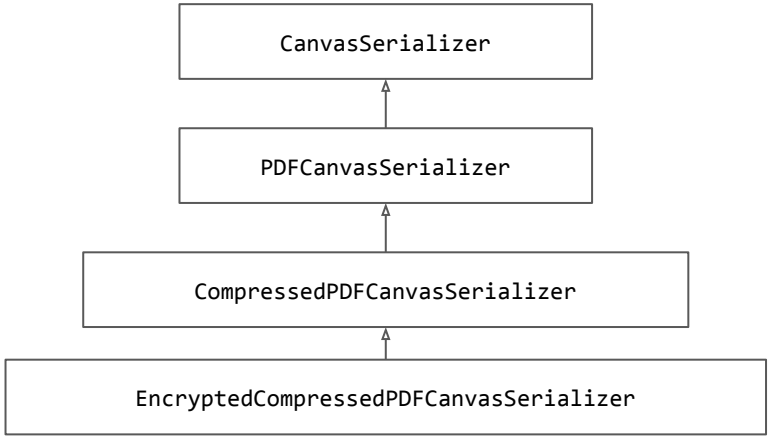
Use composition instead of inheritance

```
// Extract an interface so we can do interface inheritance instead of the implementation inheritance
abstract class CanvasSerializer:
  def serialize(canvas: Canvas): Array[Byte]

class SVGCanvasSerializer extends CanvasSerializer:
  override def serialize(canvas: Canvas): Array[Byte] = ???

class PDFCanvasSerializer extends CanvasSerializer:
  override def serialize(canvas: Canvas): Array[Byte] = ???

// And use composition to decorate the existing serializers with new behavior.
class CompressedCanvasSerializer(delegate: CanvasSerializer, level: Int) extends CanvasSerializer:
  override def serialize(canvas: Canvas): Array[Byte] =
    val bytes = delegate.serialize(canvas)
    compress(bytes)

// For example we can create this
val serializer = CompressedCanvasSerializer(PDFCanvasSerializer(), 1)
```

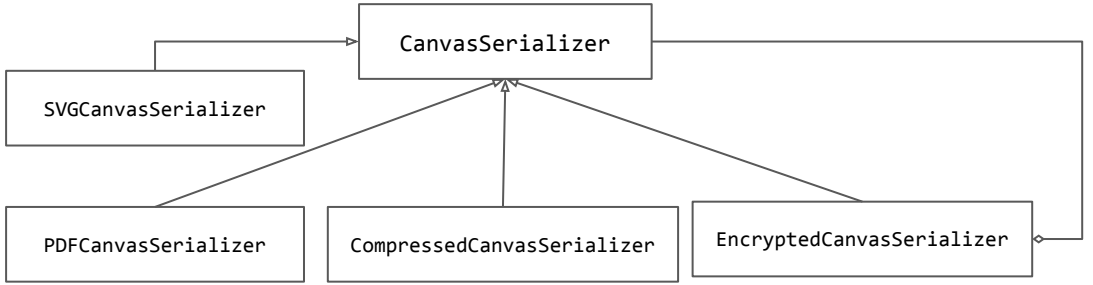
Use composition instead of inheritance

```
// The good thing is that it scales!
// we could later add other serializers, for example:
class EncryptedCanvasSerializer(delegate: CanvasSerializer, key: String) extends CanvasSerializer {
  override def serialize(canvas: Canvas): Array[Byte] =
    val bytes = delegate.serialize(canvas)
    encrypt(bytes)

// and freely combine them
val serializer =
  EncryptedCanvasSerializer(
    CompressedCanvasSerializer(
      PDFCanvasSerializer(),
      1
    ),
    "secret"
  )
```

Use composition instead of inheritance



Inheritance

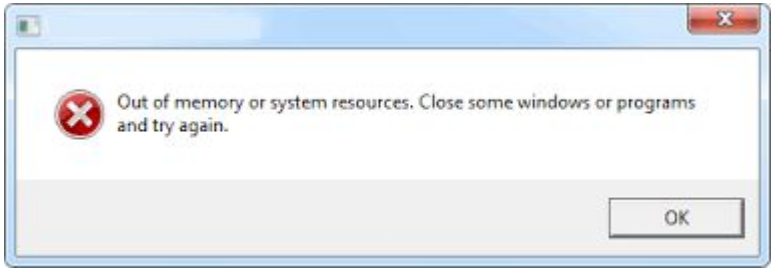


Composition

For other formats we will need to do duplicate both the compression and encryption.

Bonus Exercise

- With the `CanvasSerializer` you will soon start seeing out of memory exceptions
- Even more often when one uses compression
- Why is that?



Bonus Exercise

```
class SaveAction(serializer: CanvasSerializer):
  def run(canvas: Canvas, output: File) =
    // allocates big byte array!
    val bytes = serializer.serialize(canvas)
    // just to sink it in a file
    Files.write(output.toPath, bytes)


class CompressedCanvasSerializer(delegate: CanvasSerializer, level: Int) extends CanvasSerializer:
  override def serialize(canvas: Canvas): Array[Byte] =
    // allocates big byte array!
    val bytes = delegate.serialize(canvas)
    // compress again allocates so it can return the byte array
    compress(bytes)
```

Bonus Exercise

```
abstract class CanvasSerializer:
  def serialize(canvas: Canvas, output: OutputStream): Unit

class CompressedCanvasSerializer(delegate: CanvasSerializer, level: Int) extends CanvasSerializer:
  override def serialize(canvas: Canvas, output: OutputStream) = ???

class PDFCanvasSerializer extends CanvasSerializer:
  override def serialize(canvas: Canvas, output: OutputStream) = ???
```

```
class SaveAction(serializer: CanvasSerializer):
  def run(canvas: Canvas, output: File): Unit =
    // create the sink with the output file as target
    val fos = FileOutputStream(output)

    // add compression
    val cfos = GZIPOutputStream(fos)

    // add encryption
    val cipher = Cipher.getInstance("AES/CBC/PKCS5Padding")
    val ecfos = CipherOutputStream(cfos, cipher)

    serializer.serialize(canvas, ecfos)

    // close the stream
    fos.close()
```

```
// FIXME: this code is dangerous because
it does not handle possible errors

Wait until lecture about error handling
to see how to make it safe
```

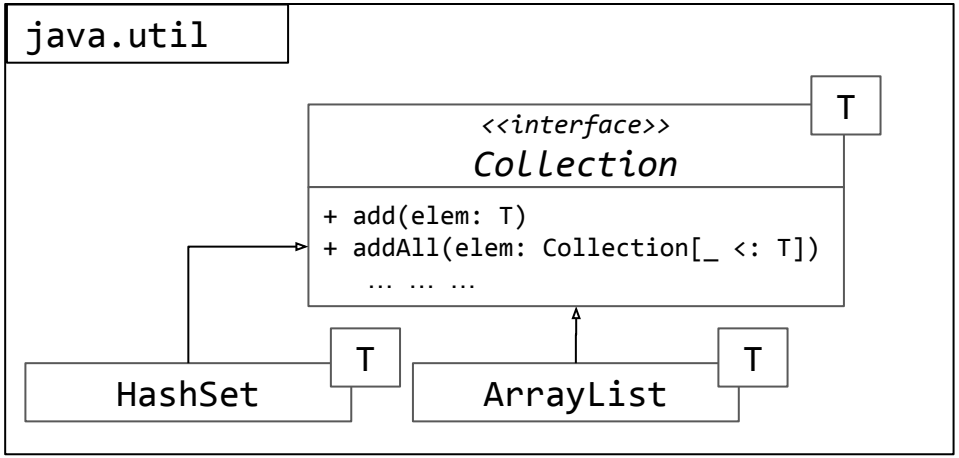
Inheritance

- Relates objects or classes together using is-a relationship
- Aims for code reuse and extension
- Allows a new object or a class to take structure and behavior of another object or class
- Enables polymorphism and code reuse

Two type of inheritance

- Implementation inheritance
  - Subclass, derived class inherits all protected and public members
    - no private (because you do not even know that they are in)
    - no overridden members (because you change the definition)
  - Need to call superclass constructor (implicitly or explicitly) to establish class invariant
    - this in turn invokes the whole constructor chain
- Interface inheritance
  - inherits only signatures
  - the new type then conforms to the interface
- Implementation inheritance is trickier than it seems

Examples



- For the following examples we use Java collections
- `java.util.Collection` is an abstract mutable unordered collection
- `java.util.ArrayList` is a concrete implementation of a mutable indexed collection backed by an array
- `java.util.HashSet` is a concrete implementation of a mutable set back by a hash table

Inheritance problems

- Create a generic Stack

```
class Stack[T] extends ArrayList[T] {
  def push(elem: T): Unit = add(elem)
  def pop(): T = remove(size() - 1)
}

val r = new Stack[Int]
r.push(1)
```

Can we do better?

- It works, but
  - the interface is fatally polluted
  - semantically, "`Stack` is an `ArrayList`" is wrong
  - inheriting from `ArrayList` breaks encapsulation
  - Stack is not a proper subtype of `ArrayList`
    - LIFO is correctly modelled by the push/pop methods
    - but not by the other methods, breaks Liskov Substitution Principle - behavioral subtyping

Inheritance problems - mitigate using composition 1

```
class Stack[T] {
  private val elems = new ArrayList[T]

  def push(elem: T): Unit = elems.add(elem)
  def pop(): T = elems.remove(size() - 1)
}

val r = new Stack[Int]
r.push(1)
```

- Use composition instead of inheritance

- Encapsulated
- Clean interface
- but, we lost the subtyping polymorphism, stack is no longer part of the collection library...

Can we do better?

Inheritance problems - mitigate using composition 2

```
class Stack[T] extends Collection[T]:
  private val elems = new ArrayList[T]

  def push(elem: T): Unit = elems.add(elem)
  def pop(): T = elems.remove(size() - 1)

  override def size() = elems.size

  override def isEmpty = elems.isEmpty

  override def contains(o: Any) = elems.contains(o)

  …
```

- Use composition instead of inheritance
- Code against interfaces
  - use interface inheritance
  - implement proper interface
  - forward methods
- Encapsulated
- Proper interface
- Keep the invariant
- Keep the subtyping

https://scastie.scala-lang.org/ldPI4reXSmCG7iLffzsd2Q

Inheritance problems

```
class CounterSet[T] extends HashSet[T]:
  private var counter = 0

  def counter: Int = counter

  override def addAll(c: Collection[_ <: T]) =
    counter += c.size()
    super.addAll(c)

  override def add(e: T) =
    counter += 1
    super.add(e)
```

- Create a HashSet that counts how many items were added[1]
- The HashSet class contains two methods for adding elements:
  - `add(elem: T)`
  - `addAll(c: Collection[_ <: T])`

```
val c = new CounterSet[Int]
c.addAll(List(1,2,3).asJava)
c.counter // ???
```

- What value will it return?

[1] J. Bloch, Effective Java, Item 18

https://scastie.scala-lang.org/ldPI4reXSmCG7iLffzsd2Q

Inheritance problems

```
class CounterSet[T] extends HashSet[T]:
  private var counter = 0

  def counter: Int = counter

  override def addAll(c: Collection[_ <: T]) =
    counter += c.size()
    super.addAll(c)

  override def add(e: T) =
    counter += 1
    super.add(e)
```

- Create a HashSet that counts how many items were added[1]
- The HashSet class contains two methods for adding elements:
  - `add(elem: T)`
  - `addAll(c: Collection[_ <: T])`

Can we do better?

```
val c = new CounterSet[Int]
c.addAll(List(1,2,3).asJava)
c.counter // 6
```

- What value will it return?

[1] J. Bloch, Effective Java, Item 18

Inheritance problems - mitigate using composition

```
class CounterSet[T] extends Set[T]:
 private val set = new HashSet[T]()
 private var counter = 0
 def counter: Int = counter

 override def addAll(c: Collection[_ <: T]) =
   counter += c.size()
   set.addAll(c)

 override def add(e: T) =
   counter += 1;
   set.add(e)

 override def size() = set.size()
```

```
val c = new CounterSet[Int]
c.addAll(List(1,2,3).asJava)
c.counter // 3
```

- The same pattern
  - use composition
  - use interface inheritance

[1] J. Bloch, Effective Java, Item 18

Composition

- Simpler relationship allowing to reuse implementation but not type
- Specifies Has-A relationship
- Often can lead to simpler and easier to maintain code

Inheritance pros and cons

- Pros
  - Implementation reuse
    - no need to repeat code that does not change
    - great for incremental changes
  - Specification reuse
    - if you understand superclass, only need to study the incremental changes subclass brings
  - Substitute new implementations
    - clients do not need to change to use subclasses
- Cons
  - Not always easy to identify the proper relationship
    - often you just need to inherit implementation, not a type
    - inheritance brings everything superclass and make it visible
  - Tight coupling with superclass
    - changes in super class can break subclass
    - subclass may rely on implementation details of superclass
  - Needs careful planning

When to use inheritance[1]

1. Both classes are in the same logical domain
2. The subclass is a proper subtype of the superclass
3. The superclass's implementation is necessary or appropriate for the subclass
4. The enhancements made by the subclass are primarily additive.

Often case in framework / plugins

[1] https://www.thoughtworks.com/insights/blog/composition-vs-inheritance-how-choose

Summary

- Prefer composition over inheritance
- Design inheritance with great care
- Code against interfaces
- Always think twice if you need really need implementation inheritance

Multiple Inheritance

Problems with Inheritance

- Single inheritance
  - Simplest model, well accepted
  - Not expressive enough
    - to factor out all the common features shared by classes in a complex hierarchy ↦ forcing code duplication
    - e.g., you have `InputStream`, `OutputStream` and want `InputOutputStream`
  - Not allowing to share features inherited from different parents
- Multiple inheritance
  - Complicated model, has not achieved wide acceptance
  - Complicated implementation
  - Ambiguity problems (the diamond problem)
  - Accessing overridden features (one `super` is not enough in presence of multiple superclasses)

Role of a Class in OOP

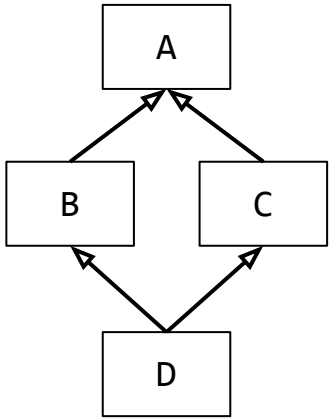
- A generator of instances
  - Provide a complete set of features
- A unit of reuse
  - Provide a minimal set of features that can be sensibly reused together
- A class[1] must have a fixed position in the class hierarchy which makes it difficult / impossible
  - to factor out wrapper methods
    - methods extending other methods with additional functionality as reusable classes
  - to resolve conflicting features inherited from different paths
  - to access or compose overridden features

[1] In a nominal statically typed world, but similar issues are in other typing schemes

Examples of approaches to multiple-inheritance

- Single inheritance with interfaces (Java, C#, Kotlin)
- Single inheritance with traits (Scala)
- Multiple inheritance (C++, Python)

The Diamond Problem



An ambiguity that arises:

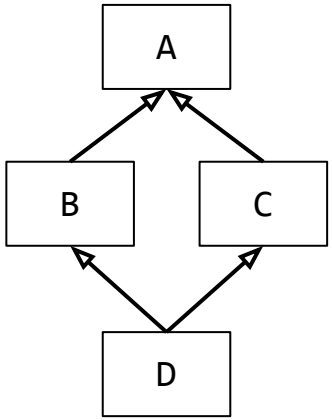
- when two classes B and C inherit from A, and
- class D inherits from both B and C.
- If there is a method in A

  that B and C have overridden,

  and D does not override,

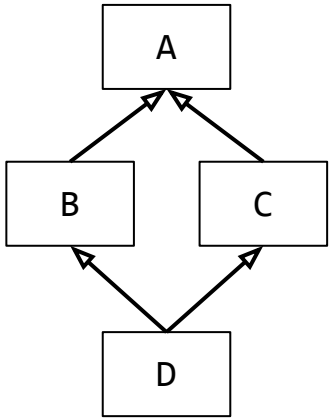
  then which version of the method does D inherit: that of B, or that of C?

The Diamond Problem in C++



- Default inheritance
  - by default each inheritance path in separated
  - D object contains two separate instances of A
  - uses of A's members have to be properly qualified
- Virtual inheritance
  - if A is inherited virtually in both B and C, only one instance of A object is created
  - if virtual and nonvirtual inheritance is mixed, there is a single virtual A, and a non-virtual A for each non-virtual inheritance path to A
  - C++ requires stating explicitly which parent class should be used

The Diamond Problem in Python

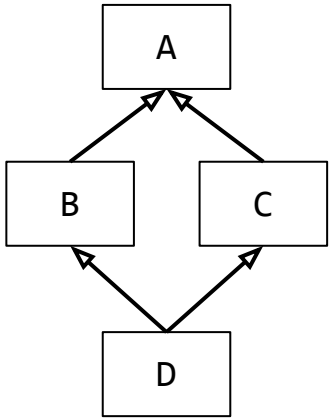



```
class A: pass

class B(A): pass

class C(A): pass

class D(B, C): pass
```

- Creates a list of classes using the C3 linearization:
  - children precede their parents, and
  - if a class inherits from multiple classes, they are kept in the order specified
- E.g.: the method resolution order (MRO) is: D, B, C, A

```
- >>> D.__mro__
  (<class '__main__.D'>,
   <class '__main__.B'>,
   <class '__main__.C'>,
   <class '__main__.A'>,
   <type 'object'>)
```

The Diamond Problem in Java?



- Until Java 8 no problems
- Java 8 brought default methods
  - allowing one to add implementation to traits
  - this can cause diamond problem
  - must be resolved manually using explicit override of the conflicting method

The Diamond Problem in Java?

```
interface A {
    default void m() {
        System.out.println("A.m");
    }
}

interface B extends A {
    default void m() {
        System.out.println("B.m");
    }
}

interface C extends A {
    default void m() {
        System.out.println("C.m");
    }
}

class D implements B, C {
}
```

```
class D implements B, C {
    // The conflict has to be manually resolved
    @Override
    public void m() {
        B.super.m();
    }
}
```

```
error: types B and C are incompatible;
class D implements B, C {
^
  class D inherits unrelated defaults for m() from types B and C
1 error
```

Traits

Traits

- Fundamental units of reuse (that can be divorced from any class hierarchy)
- Contains method and field
  - Provide methods & fields
  - Require methods & fields (that are used but not defined in the trait)
- A class can mix in any number of traits and conflicts are resolved using overriding
- Classes retain primary roles as instance generators
- A trait is almost like an abstract class, but
  - cannot create instances
  - the calls to `super` are resolved dynamically (unlike in a class where it is static)
- Two common patterns
  - Widen thin interfaces
  - Stackable modifications

Widen thin interfaces

- Client/implementer trade-off between rich / thin interfaces
  - rich interfaces are convenient for clients (more methods to pick from)
  - thin interfaces are convenient for implementers (less code to write)

- For example, think about the difference between
  - using Iterator vs
  - creating a new Iterator implementation

Widen thin interfaces

```
class Rectangle(val topLeft: Point,
                val bottomRight: Point):
  def left = topLeft.x
  def right = bottomRight.x
  def top = topLeft.y
  def bottom = bottomRight.y
  def width = right - left
  def height = bottom - top
  ...
```

unrelated classes
(yet, with similar behavior)

```
abstract class UIComponent:
 def topLeft: Point
 def bottomRight: Point
 def left = topLeft.x
 def right = bottomRight.x
 def top = topLeft.y
 def bottom = bottomRight.y
 def width = right - left
 def height = bottom - top
 ...
```

```
trait Rectangular:
  def topLeft: Point
  def bottomRight: Point

  def left = topLeft.x
  def right = bottomRight.x
  def top = topLeft.y
  def bottom = bottomRight.y
  def width = right - left
  def height = bottom - top
```

required methods
(It can only be mixed in to a type that has these methods)

provided methods

```
class Rectangle(val topLeft: Point,
                val bottomRight: Point) extends Rectangular:
  ...
}

abstract class UIComponent extends Rectangular
  ...
}
```

trait mix in

Example of CharSequence

java.lang

Interface CharSequence

All Known Subinterfaces:
Name

All Known Implementing Classes:
CharBuffer, Segment, String, StringBuffer, StringBuilder

---

public interface **CharSequence**

A CharSequence is a readable sequence of char values. This interface provides uniform, read-only access to many different kinds of char sequences. A char value represents a character in the *Basic Multilingual Plane (BMP)* or a surrogate. Refer to Unicode Character Representation for details.

This interface does not refine the general contracts of the equals and hashCode methods. The result of comparing two objects that implement CharSequence is therefore, in general, undefined. Each object may be implemented by a different class, and there is no guarantee that each class will be capable of testing its instances for equality with those of the other. It is therefore inappropriate to use arbitrary CharSequence instances as elements in a set or as keys in a map.

Since:
1.4

Method Summary

All Methods | Instance Methods | Abstract Methods | Default Methods

Modifier and Type	Method and Description
char	**charAt**(int index) Returns the char value at the specified index.
default IntStream	**chars**() Returns a stream of int zero-extending the char values from this sequence.
default IntStream	**codePoints**() Returns a stream of code point values from this sequence.
int	**length**() Returns the length of this character sequence.
CharSequence	**subSequence**(int start, int end) Returns a CharSequence that is a subsequence of this sequence.
String	**toString**() Returns a string containing the characters in this sequence in the same order as this sequence.

https://docs.oracle.com/javase/8/docs/api/java/lang/CharSequence.html

https://www.scala-lang.org/api/3.5.2/scala/collection/StringOps.html

Exercise - Comparison operators for `Rational` class

```
final class Rational(numer: Int, denom: Int):
  …

def f(x: Rational, y: Rational, z: Rational) = y >= x && y <= z
```

```
class Rational(numer: Int, denom: Int):
  def < (that: Rational) =
    this.numer * that.denom < that.numer * this.denom

  def > (that: Rational) = that < this
  def <= (that: Rational) = (this < that) || (this == that)
  def >= (that: Rational) = (this > that) || (this == that)
```

Exercise - Comparison operators for `Rational` class

```
final class Rational(numer: Int, denom: Int) extends Ordered[Rational]
  ...
  def compare(that: Rational) =
   (this.numer * that.denom) - (that.numer * this.denom)
```

trait mix in

```
trait Ordered[A] extends Any with java.lang.Comparable[A]:

 def compare(that: A): Int

  def <  (that: A): Boolean = (this compare that) <  0

  def >  (that: A): Boolean = (this compare that) >  0

  def <= (that: A): Boolean = (this compare that) <= 0

  def >= (that: A): Boolean = (this compare that) >= 0

  def compareTo(that: A): Int = compare(that)
```

required methods
(It can only be mixed in to a type that has these methods)

provided methods

Stackable pattern

- Traits can represent modifications (essentially a decorator)
- The can be stacked next to each other

Stackable pattern - example

Create a stack of integers

Stackable pattern - example

```
trait IntStack:
  def push(e: Int): IntStack
  def pop(): Int

class BasicIntStack extends IntStack:
  private var elems = List[Int]()

  def push(e: Int): IntStack =
    elems = e :: elems
    this

  def pop(): Int = elems match
    case x :: xs =>
      elems = xs
      x
    case Nil => throw new NoSuchElementException

  override def toString = elems.mkString("Stack(", ",", ")")
```

Stackable pattern - example

Create a stack of integers

Create a stack of integers that increments pushed integers

Create a stack of integers that doubles pushed integers

Create a stack of integers that filters out negative values

Create a stack of integers that can increment, double and filter out values

Stackable pattern - example

```
trait IntStack:
  def push(e: Int): IntStack
  def pop(): Int
```

This type can only be mixed into classes that mix in IntStack

```
trait Incrementing extends IntStack:
  abstract override def push(e: Int): IntStack = super.push(e + 1)

trait Doubling extends IntStack:
  abstract override def push(e: Int): IntStack = super.push(e * 2)

trait FilteringNegative extends IntStack:
  abstract override def push(e: Int): IntStack = if (e >= 0) super.push(e) else this
```

This trait must be mixed into some class that has a concrete definition of the method

https://scastie.scala-lang.org/9mqx2LCiQwafyqQiBNOKsg

Stackable pattern - example

```
trait IntStack
class BasicIntStack extends IntStack
trait Incrementing extends IntStack
trait Doubling extends IntStack
trait FilteringNegative extends IntStack


class MyStack extends BasicIntStack with Doubling with Incrementing with FilteringNegative
val s1 = new MyStack
val s2 = new BasicIntStack() with Incrementing with Doubling

trait Filtering(p: Int => Boolean) extends IntStack:
  abstract override def push(e: Int): IntStack = if (p(e)) super.push(e) else this

val s3 = new MyStack() with Filtering(x => (x & 1) == 0)
s3.push(1)
s3.push(2)
s3.push(3)
s3 // ???    6
```

Ad hoc mix in

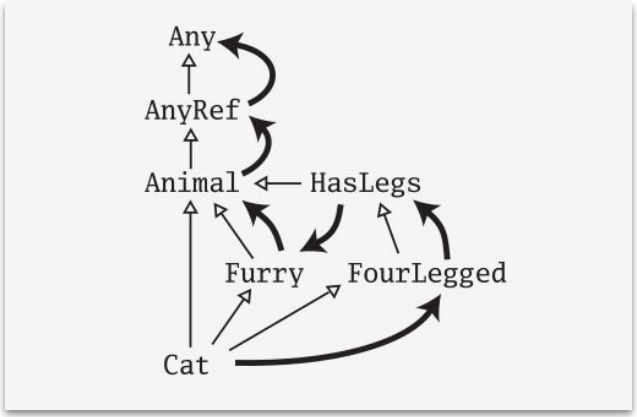
Traits vs Multiple Inheritance

- In traits `super` uses late binding - dispatched dynamically
- Method call is determined by linearization

```
trait Incrementing extends IntStack {
  abstract override def push(e: Int): IntStack = super.push(e + 1)
}

trait Doubling extends IntStack {
  abstract override def push(e: Int): IntStack = super.push(e * 2)
}


val s1 = new BasicIntStack() with Incrementing with Doubling
s1.push(2)
s1.pop()

val s2 = new BasicIntStack() with Doubling with Incrementing
s2.push(2)
s2.pop()
```

Traits Linearization

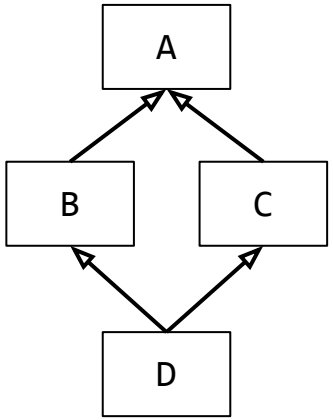
```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```



- Method are resolved using a right-first depth-first search of extended 'traits', before eliminating all but the last occurrence of each module in the resulting list

```
Animal:     Animal -> AnyRef -> Any
Furry:      Furry -> Animal -> AnyRef -> Any
FourLegged: HasLegs -> Furry -> Animal -> AnyRef -> Any
HasLegs:    HasLegs -> Animal -> AnyRef -> Any
Cat:        Cat -> FourLegged -> HasLegs -> Furry -> Animal -> AnyRef -> Any
```

The Diamond Problem in Scala?



- Scala allows multiple inheritance through traits
- It resolves method through linearization using right-first, depth-first algorithm, no diamond problem

```
trait A:
  def m(): Unit = println("A.m()")

trait B extends A:
    override def m(): Unit = println("B.m()")

trait C extends A:
    override def m(): Unit = println("C.m()")

class D extends B, C

def main(args: Array[String]): Unit =
    val d = D()
    d.m() // ???
```

The Diamond Problem in Scala?

```scala
trait B:
  def m(): Unit = println("B.m()")

trait C:
  def m(): Unit = println("C.m()")

class D extends B, C

def main(args: Array[String]): Unit =
    val d = D()
    d.m() // ???
```

- No diamond

```
[error] error overriding method m in trait B of type (): Unit;
[error]   method m in trait C of type (): Unit class D inherits
conflicting members:
[error]   method m in trait B of type (): Unit  and
[error]   method m in trait C of type (): Unit
[error] (Note: this can be resolved by declaring an override in class D.)
[error] class D extends B,C
[error]       ^
```

The Diamond Problem in Scala?

```
trait B:
  def m(): Unit = println("B.m()")

trait C:
  def m(): Unit = println("C.m()")

class D extends B, C:
  override def m(): Unit = super.m()

def main(args: Array[String]): Unit =
    val d = D()
    d.m() // C.m()
```

- No diamond
- Still there can be a conflict that needs an explicit resolution

The Diamond Problem in Scala?

```
trait B:
  def m(): Unit = println("B.m()")

trait C:
  def m(): Unit = println("C.m()")

class D extends B, C:
  override def m(): Unit = super[B].m()

def main(args: Array[String]): Unit =
    val d = D()
    d.m() // B.m()
```

- No diamond
- Still there can be a conflict that needs an explicit resolution
- Scala supports qualified `super`

To Trait or not to Trait

- If the behavior will not be reused: a concrete class
- If it might be reused in multiple, unrelated classes: a trait
- If you want to inherit from it in Java code: an abstract class
- If you still do not know: a trait (keeps more options open)

Why not multiple inheritance?

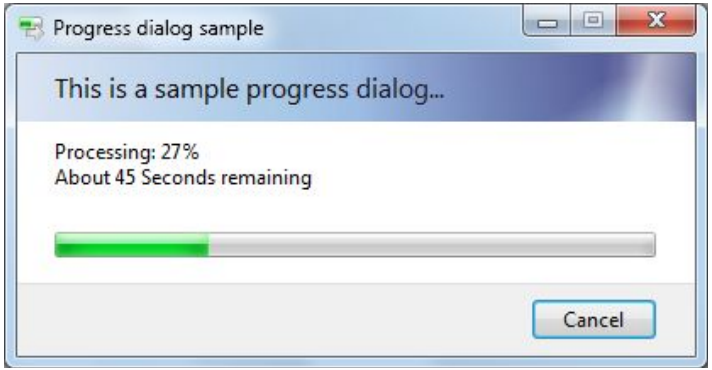
- the interpretation of super in traits is dynamic
  - the method called is determined by a linearization of the classes and traits that are mixed into a class

```
// Multiple inheritance thought experiment
val s = new BasicIntStack with Incrementing
                          with Doubling
s.push(42) // which put would be called?
           // the first one, i.e., Incrementing?
           // the last one, i.e., Doubling?
```

```
// perhaps we could resolve it manually
trait MyStack extends BasicIntStack with Incrementing
                                     with Doubling {
  def push(x: Int) = {
    super[Incrementing].push(x)
    super[Doubling].push(x)
    // in such a case:
    // x will be pushed twice
    // none will be doubled and incremented
    // at the same time
  }
}
```

Bonus Exercise 2

- Saving could take a lot of time
- Add a progress bar to visualize the progress

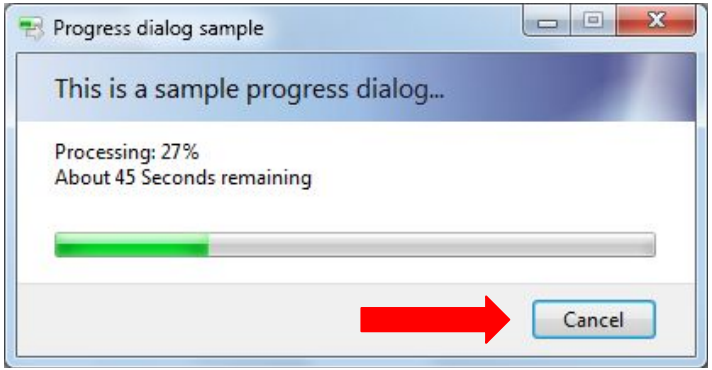


Bonus Exercise 2

```
trait ProgressMonitor:
  def begin(totalWork: Int): Unit
  def progress(worked: Int): Unit
  def done(): Unit


class PDFCanvasSerializer extends CanvasSerializer:
  def serialize(canvas: Canvas, output: OutputStream, monitor: ProgressMonitor): Unit =
    monitor.begin(canvas.size)

    canvas.shapes.foreach { s =>
      serializeOne(s, output)
      monitor.progress(1)
    }

    monitor.done()

  def serializeOne(shape: Shape, output: OutputStream): Unit


class NullProgressMonitor extends ProgressMonitor:
  def begin(totalWork: Int): Unit = ()
  def progress(worked: Int): Unit = ()
  def done(): Unit

class PdfCanvasSerializer:
  def serialize(canvas: Canvas, output: OutputStream, monitor: ProgressMonitor = NullProgressMonitor): Unit = …
```

Bonus Exercise 2

- Saving could take a lot of time
- Add a progress bar to visualize the progress



- What about the cancel button?

Bonus Exercise 2

```
trait ProgressMonitor:
  def begin(totalWork: Int): Unit
  def progress(worked: Int): Unit
  def done(): Unit
  def isCancelled: Boolean


class PDFCanvasSerializer extends CanvasSerializer:
  def serialize(canvas: Canvas, output: OutputStream, monitor: ProgressMonitor): Unit =
    monitor.begin(canvas.size)

    canvas.shapes.takeWhile(_ => !monitor.isCancelled).foreach{ s =>
      serializeOne(s, output)
      monitor.progress(1)
    }

    monitor.done()

  def serializeOne(shape: Shape, output: OutputStream): Unit
```

Class hierarchies

Exercise

- What is the type of `x`?

```
val x = if (true) 1 else "Huh?"
```

- Need to find a common super type for both `Int` and `String`

```
val x: Any = if (true) 1 else "Huh?"
```

Exercise

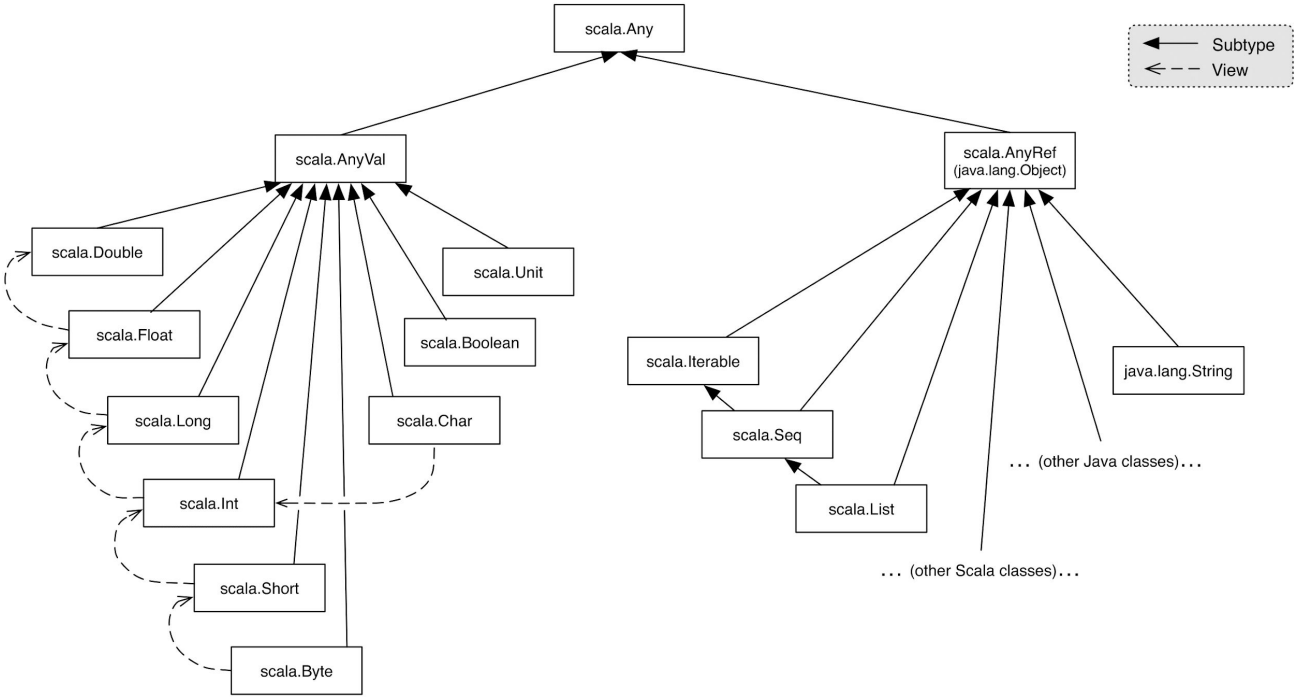
- What is the type of `x`?

```
val x = if (false) 1
```

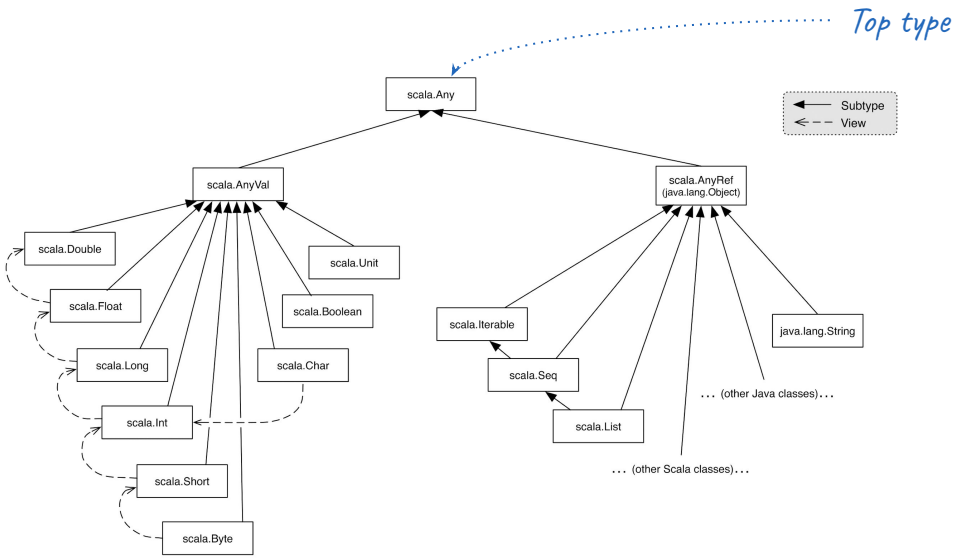
- The missing else branch will have type `Unit`
- Need to find a common super type for both `Int` and `Unit`

```
val x: AnyVal = if (false) 1
```

Scala Class hierarchy



Scala Class hierarchy



Top types

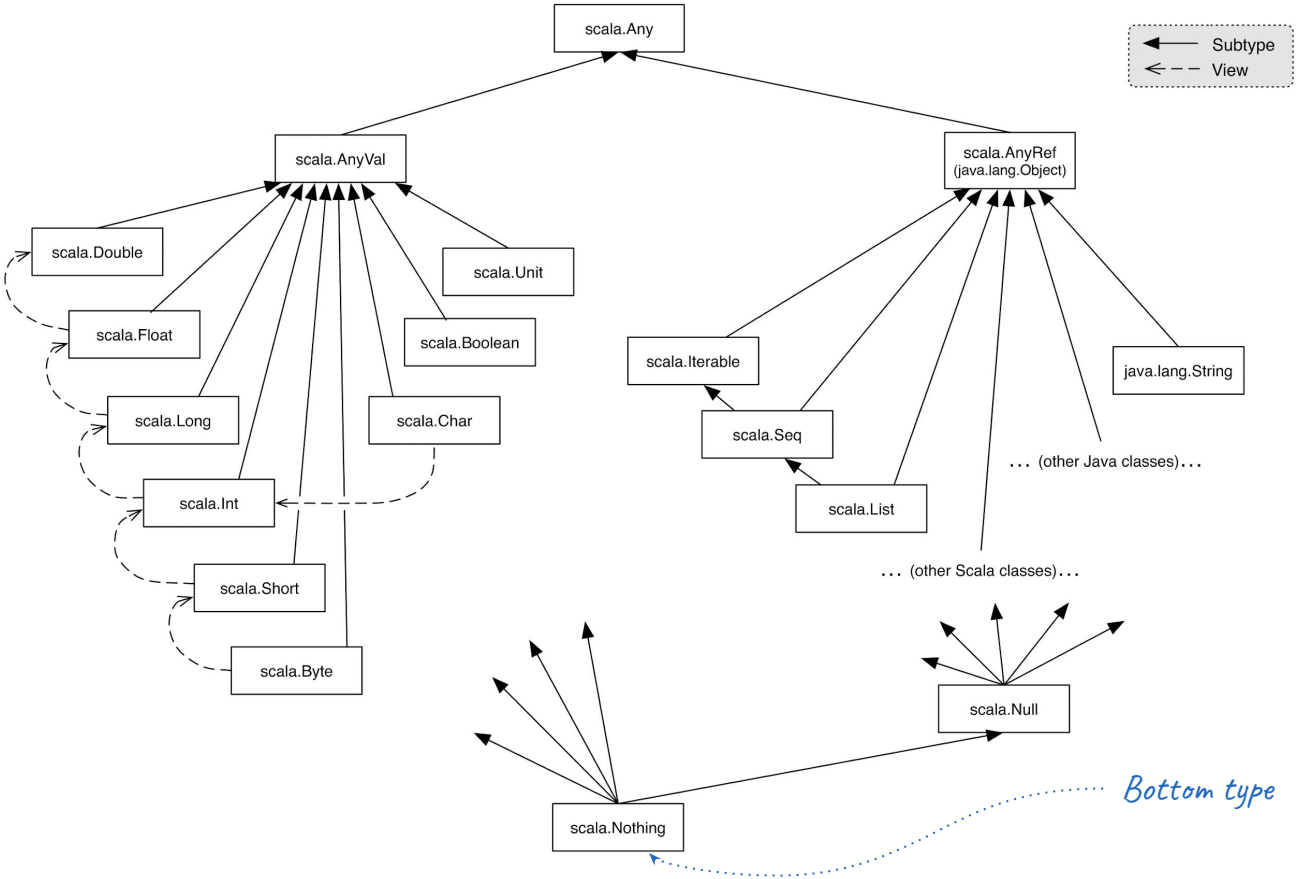
- `scala.Any`
  - base type of all types
  - `==, !=, equals, hashCode`
  - `toString`
- `scala.AnyVal`
  - base type of all primitive types
  - alias java.lang.Object
- `scala.AnyRef`
  - base type of all reference types

Exercise

- What is the type of `Nil.head` and `Nil.tail`?

```
trait IntList:
  def head: Int
  def tail: IntList
  ...

class Nil extends IntList:
  def head = sys.error("Empty.head")
  def tail = sys.error("Empty.tail")
  ...
```

Complete Scala Class hierarchy



Nothing type

- At bottom of the hierarchy
- Subclass of any every other type
- Does not have a value

Why is it uself?

- Signal abnormal termination
- Element of an empty collection

```
def f(x: Int): String = ???
```

```
/** `???` can be used for marking methods that
 * remain to be implemented.
 *
 *  @throws NotImplementedError when `???` is
 *  invoked.
 *  @group utilities
 */
def ??? : Nothing = throw new NotImplementedError
```

scala.Predef

Null type

- Every reference class has `null` reference
- Type of `null` is `Null`
- Subtype of every other reference type
  - Subclass of `java.lang.Object` (`scala.AnyRef`)
  - Not compatible with `AnyVal`

```
val x = null
val s: String = null
val i: Int = null      // compile-time exception
```

Class hierarchies - imports

- In class-based OOP programs are organized in classes (in Scala: Class, Object or Trait)
- Classes themselves are organized in some units (in Scala: packages)
- There is an import mechanism that allow one to import definitions from various modules

```
package p1

class A:
  def b = 1

object B:
  def f1 = ???
  def f2 = ???

object C
```

```
package p2

import p1.A
import p1.{A, C}

import p1._


import p1.B.{f1, f2}
import p1.B.{f1=g}

import p1.B._
```

Name imports

Wildcard imports

Name imports

Wildcard imports

```
val a = A()

// import simply brings
// symbols into its current
// lexical scope
import a._

println(b) // prints 1
```