Varieties of abstraction

Abstraction over computation: procedures (methods)
int x1, y1, x2, y2;
Math.sqrt(x1*x1 + y1*y1);
Math.sqrt(x2*x2 + y2*y2);

Abstraction over data: ADTs (classes, interfaces)
Point p1, p2;

Abstraction over types: polymorphism (generics)
Point<Integer>, Point<Double>

Why we love abstraction

Hide details
– Avoid distraction
– Permit details to change later

Give a meaningful name to a concept

Permit reuse in new contexts
– Avoid duplication: error-prone, confusing
– Save reimplementation effort
– Helps to “Don’t Repeat Yourself”

Related abstractions

interface ListOfNumbers {
    boolean add(Number elt);
    Number get(int index);
}

interface ListOfIntegers {
    boolean add(Integer elt);
    Integer get(int index);
}

… and many, many more

// abstracts over element type
interface List<E> {
    boolean add(E n);
    E get(int index);
}

An analogous parameter

interface ListOfIntegers {
    boolean add(Integer elt);
    Integer get(int index);
}

interface List<E> {
    boolean add(E n);
    E get(int index);
}

• Declares a new variable, called a formal parameter
• Instantiate with any expression of the right type
  • E.g., lst.add(7)
• Type of add is Integer → boolean

• Declares a new type variable, called a type parameter
• Instantiate with any (reference) type
  • E.g., List<String>
• “Type” of List is Type → Type
• Never just use List (in Java for backward-compatibility)

Type variables are types

class NewSet<T> implements Set<T> {
    // rep invariant:
    // non-null, contains no duplicates
    // ...
    List<T> theRep;
    T lastItemInserted;
    …
}
Declaring and instantiating generics

class Name<TypeVar1, …, TypeVarN> {...}
interface Name<TypeVar1, …, TypeVarN> {...}

– Convention: One-letter name such as:
  T for Type, E for Element,
  K for Key, V for Value, ...

To instantiate a generic class/interface, client supplies type arguments:
Name<Type1, …, TypeN>

Restricting instantiations by clients

boolean add1(Object elt);
boolean add2(Number elt);
add1(new Date()); // OK
add2(new Date()); // compile-time error

interface List1<E extends Object> {...}
interface List2<E extends Number> {...}
List1<Date> // OK, Date is a subtype of Object
List2<Date> // compile-time error, Date is not a // subtype of Number

Using type variables

Code can perform any operation permitted by the bound
– Because we know all instantiations will be subtypes!
– An enforced precondition on type instantiations

interface List1<E extends Object> {
    void m(E arg) {
        arg.asInt();  // compiler error, E might not // support asInt
    }
}

interface List2<E extends Number> {
    void m(E arg) {
        arg.asInt();  // OK, since Number and its // subtypes support asInt
    }
}

More examples

public class Graph<N> implements Iterable<N> {
    private final Map<N, Set<N>> node2neighbors;
    public Graph(Set<N> nodes, Set<Tuple<N,N>> edges){
        ...
    }
}

public interface Path<N, P extends Path<N,P>>
    extends Iterable<N>, Comparable<Path<?, ?>> {
    public Iterator<N> iterator();
    ...
}

More bounds

<TypeVar extends SuperType>
– An upper bound; accepts given supertype or any of its subtypes

<TypeVar extends ClassA & InterfaceB & InterfaceC & …>
– Multiple upper bounds (superclass/interfaces) with &

<TypeVar super SubType>
– A lower bound; accepts the given subtype or any of its supertypes

Example:
// tree set works for any comparable type
public class TreeSet<T extends Comparable<T>> {
    ...
}
Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of bounding generics

- Now:
  - Generic methods [not just using type parameters of class]
  - Generics and subtyping
  - Using bounds for more flexible subtyping
  - Using wildcards for more convenient bounds
  - Related digression: Java’s array subtyping
  - Java realities: type erasure
    - Unchecked casts
    - equals interactions
    - Creating generic arrays

Not all generics are for collections

```java
class Utils {
    static double sumList(List<Number> lst) {
        double result = 0.0;
        for (Number n : lst) {
            result += n.doubleValue();
        }
        return result;
    }
    static Number choose(List<Number> lst) {
        int i = ... // random number < lst.size
        return lst.get(i);
    }
}
```

Weaknesses

- Would like to use `sumList` for any subtype of `Number`
  - For example, `Double` or `Integer`
  - But as we will see, `List<Double>` is not a subtype of `List<Number>`

- Would like to use `choose` for any element type
  - i.e., any subclass of `Object`
  - No need to restrict to subclasses of `Number`
  - Want to tell clients more about return type than `Object`

- `Utils` is not generic, the methods should be generic

Using generics in methods

- Instance methods can use type parameters of the class

- Instance methods and static methods can have their own type parameters
  - Generic methods

- Callers to generic methods need not explicitly instantiate the methods’ type parameters
  - Compiler just figures it out for you
  - Type inference

More examples

```java
<T extends Comparable<T>> T max(Collection<T> c) {
    ...
}

<T extends Comparable<T>>
void sort(List<T> list) {
    // ... use list.get() and T’s compareTo
}
```

(Have to declare type parameter(s))

```java
<T> void copyTo(List<T> dst, List<T> src) {
    for (T t : src)
        dst.add(t);
}
```

(Have to declare type parameter(s))

(This one “works” but will make it even more useful later by adding more bounds)
Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of bounding generics
- Now:
  - Generic methods [not just using type parameters of class]
  - Generics and subtyping
  - Using bounds for more flexible subtyping
  - Using wildcards for more convenient bounds
  - Related digression: Java’s array subtyping
  - Java realities: type erasure
    -Unchecked casts
    - equals interactions
    - Creating generic arrays

Generics and subtyping

- Integer is a subtype of Number
- Is List<Integer> a subtype of List<Number>?
- Use subtyping rules (stronger, weaker) to find out…

List<Number> and List<Integer>

```java
interface List<T> {
    boolean add(T elt);
    T get(int index);
}
```

So type List<Number> has:
- boolean add(Number elt);
- Number get(int index);
So type List<Integer> has:
- boolean add(Integer elt);
- Integer get(int index);

Java subtyping is invariant with respect to generics
- Not covariant and not contravariant
- Neither List<Number> nor List<Integer> subtype of other

Hard to remember?

If Type2 and Type3 are different, then Type1<Type2> is not a subtype of Type1<Type3>

Previous example shows why:
- Observer method prevents “one direction”
- Mutator/producer method prevents “the other direction”

If our types have only observers or only mutators, then one direction of subtyping would be sound
- But Java’s type system does not “notice this” so such subtyping is never allowed in Java

Read-only allows covariance

```java
interface List<T> {
    T get(int index);
}
```

So type List<Number> has:
- Number get(int index);
So type List<Integer> has:
- Integer get(int index);
So covariant subtyping would be correct:
- List<Integer> a subtype of List<Number>

But Java does not analyze interface definitions like this
- Conservatively disallows this subtyping

Write-only allows contravariance

```java
interface List<T> {
    boolean add(T elt);
}
```

So type List<Number> has:
- boolean add(Number elt);
So type List<Integer> has:
- boolean add(Integer elt);
So contravariant subtyping would be correct:
- List<Number> a subtype of List<Integer>

But Java does not analyze interface definitions like this
- Conservatively disallows this subtyping
About the parameters

- So we have seen `List<Integer>` and `List<Number>` are not subtype-related
- But there is subtyping “as expected” on the generic types themselves
- Example: If `HeftyBag` extends `Bag`, then
  - `HeftyBag<Integer>` is a subtype of `Bag<Integer>`
  - `HeftyBag<Number>` is a subtype of `Bag<Number>`
  - `HeftyBag<String>` is a subtype of `Bag<String>`
  - ...

Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of bounding generics
- Now:
  - Generic methods [not just using type parameters of class]
  - Generics and subtyping
  - Using bounds for more flexible subtyping
  - Using wildcards for more convenient bounds
  - Related digression: Java’s array subtyping
  - Java realities: type erasure
    - Unchecked casts
    - `equals` interactions
    - Creating generic arrays

More verbose first

Now:
- How to use `type bounds` to write reusable code despite invariant subtyping
- Elegant technique using generic methods
- General guidelines for making code as reusable as possible

Then: Java wildcards
- Essentially provide the same expressiveness
- `Less verbose`: No need to declare type parameters that would be used only once
- `Better style` because Java programmers recognize how wildcards are used for common idioms
  - Easier to read (?) once you get used to it

Best type for `addAll`

```java
interface Set<E> {
  // Adds all elements in c to this set
  // (that are not already present)
  void addAll(_______ c);
}
```

What is the best type for `addAll`’s parameter?
- Allow as many clients as possible...
- … while allowing correct implementations

Best type for `addAll`

```java
interface Set<E> {
  // Adds all elements in c to this set
  // (that are not already present)
  void addAll(_______ c);
}
```

```java
void addAll(Set<E> c);
```

Too restrictive:
- Does not let clients pass other collections, like `List<E>`
- Better: use a supertype interface with just what `addAll` needs
- This is not related to invariant subtyping [yet]

Best type for `addAll`

```java
interface Set<E> {
  // Adds all elements in c to this set
  // (that are not already present)
  void addAll(_______ c);
}
```

```java
void addAll(Collection<E> c);
```

Too restrictive:
- Client cannot pass a `List<Integer>` to `addAll` for a `Set<Number>`
- Should be okay because `addAll` implementations only need to read from `c`, not put elements in it
- This is the invariant-subtyping limitation
Best type for `addAll`

```java
interface Set<E> {
    // Adds all elements in c to this set
    // (that are not already present)
    void addAll(_______ c);
}

<T extends E> void addAll(Collection<T> c);
```

The fix: A bounded generic type parameter
- Now client can pass a `List<Integer>` to `addAll` for a `Set<Number>`
- `addAll` implementations won’t know what element type `T` is, but will know it is a subtype of `E`
  - So it cannot add anything to collection `c` refers to
  - But this is enough to implement `addAll`

Revisit copy method

Earlier we saw this:
```java
<T> void copyTo(List<T> dst, List<T> src) {
    for (T t : src)
        dst.add(t);
}
```

Now see this is more useful to clients:
```java
<T1, T2 extends T1> void copyTo(List<T1> dst, List<T2> src) {
    for (T2 t : src)
        dst.add(t);
}
```

Where are we?
- Done:
  - Basics of generic types for classes and interfaces
  - Basics of bounding generics
- Now:
  - Generic methods [not just using type parameters of class]
  - Generics and subtyping
  - Using bounds for more flexible subtyping
  - Using wildcards for more convenient bounds
  - Related digression: Java’s array subtyping
  - Java realities: type erasure
    - Unchecked casts
    - `equals` interactions
    - Creating generic arrays

Wildcards

Syntax: For a type-parameter instantiation (inside the `<…>), can write:
- `? extends Type`, some unspecified subtype of `Type`
- `?`, shorthand for `? extends Object`
- `? super Type`, some unspecified supertype of `Type`

A wildcard is essentially an anonymous type variable
- Each ? stands for some possibly-different unknown type
- Use a wildcard when you would use a type variable exactly once, so no need to give it a name
- Avoids declaring generic type variables
- Communicates to readers of your code that the type’s “identity” is not needed anywhere else

Examples

[Compare to earlier versions using explicit generic types]
```java
interface Set<E> {
    void addAll(Collection<? extends E> c);
}
```
- More flexible than `void addAll(Collection<E> c);`
- More idiomatic (but equally powerful) to
```java
<T extends E> void addAll(Collection<T> c);
```

More examples

```java
<T extends Comparable<? super T>> T max(Collection<T> c);  // No change because `T` used more than once
<T> void copy(List<? super T> dst, List<? extends T> src);
```

Why this “works?”
- Lower bound of `T` for where callee puts values
- Upper bound of `T` for where callee gets values
- Callers get the subtyping they want
  - Example: `copy(numberList, integerList)`
  - Example: `copy(stringList, stringList)`
PECS: Producer Extends, Consumer Super

Where should you insert wildcards?

Should you use `extends` or `super` or neither?

- Use `extends T` when you get values (from a producer)
  - No problem if it's a subtype
- Use `super T` when you put values (into a consumer)
  - No problem if it's a supertype
- Use neither (just `T`, not ?) if you both get and put

```java
<T> void copy(List<? super T> dst, List<? extends T> src);
```

? versus Object

? indicates a particular but unknown type

```java
void printAll(List<??> lst) {...}
```

Difference between List<?> and List<Object>:

- Can instantiate ? with any type: Object, String, ...
- List<Object> is restrictive; wouldn't take a List<String>

Difference between List<Foo> and List<? extends Foo>

- In latter, element type is one unknown subtype of Foo
  - Example: List<? extends Animal> might store only Giraffes but not Zebras
- Former allows anything that is a subtype of Foo in the same list
  - Example: List<Animal> could store Giraffe and Zebras

Legal operations on wildcard types

```java
Object o;
Number n;
Integer i;
PositiveInteger p;
List<? extends Integer> lei;
```

First, which of these is legal?

```java
lei = new ArrayList<Object>;
lei = new ArrayList<Number>;
lei = new ArrayList<Integer>;
lei = new ArrayList<PositiveInteger>;
lei = new ArrayList<NegativeInteger>;
```

```java
lsi = new ArrayList<Object>;
lsi = new ArrayList<Number>;
lsi = new ArrayList<Integer>;
lsi = new ArrayList<PositiveInteger>;
lsi = new ArrayList<NegativeInteger>;
```

Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of bounding generics
- Now:
  - Generic methods [not just using type parameters of class]
  - Generics and subtyping
  - Using bounds for more flexible subtyping
  - Using wildcards for more convenient bounds
  - Related digression: Java's array subtyping
  - Java realities: type erasure
    - Unchecked casts
    - equals interactions
  - Creating generic arrays

Java arrays

We know how to use arrays:

- Declare an array holding Type elements: Type[]
- Get an element: x[i]
- Set an element: x[i] = e;

Java included the syntax above because it's common and concise

But can reason about how it should work the same as this:

```java
class Array<T> {
    public T get(int i) { ... "magic" ... }
    public T set(T newVal, int i) { ... "magic" ... }
}
```

So: If Type1 is a subtype of Type2, how should Type1[] and Type2[] be related??
Surprise!

- Given everything we have learned, if Type1 is a subtype of Type2, then Type1[] and Type2[] should be unrelated
  - Invariant subtyping for generics
  - Because arrays are mutable
- But in Java, if Type1 is a subtype of Type2, then Type1[] is a subtype of Type2[]
  - Not true subtyping: the subtype does not support setting an array index to hold a Type2
  - Java (and C#) made this decision in pre-generics days
    - Else cannot write reusable sorting routines, etc.
    - Now programmers are used to this too-lenient subtyping

What can happen: the good

Programmers can use this subtyping to “do okay stuff”

```java
void maybeSwap(LibraryHolding[] arr) {
  if(arr[17].dueDate() < arr[34].dueDate())
    // ... swap arr[17] and arr[34]
}
```

// client with subtype
Book[] books = ...;
maybeSwap(books); // relies on covariant
// array subtyping

What can happen: the bad

Something in here must go wrong!

```java
void replace17(LibraryHolding[] arr, LibraryHolding h) {
  arr[17] = h;
}
```

// client with subtype
Book[] books = ...;
LibraryHolding theWall = new CD("Pink Floyd", "The Wall", …);
replace17(books, theWall);
Book b = books[17]; // would hold a CD
b.getChapters(); // so this would fail

Java’s choice

- Recall Java’s guarantee: Run-time type is a subtype of the compile-time type
  - This was violated for the Book b variable
- To preserve the guarantee, Java would never get that far:
  - Each array “knows” its actual run-time type (e.g., Book [])
  - Trying to store a (run-time) supertype into an index causes ArrayStoreException
- So the body of replace17 would raise an exception
  - Even though replace17 is entirely reasonable
    - And fine for plenty of “careful” clients
  - Every Java array-update includes this run-time check
    - (Array-reads never fail this way – why?)
  - Beware array subtyping!

Where are we?

- Done:
  - Basics of generic types for classes and interfaces
  - Basics of bounding generics
- Now:
  - Generic methods [not just using type parameters of class]
    - Generics and subtyping
    - Using bounds for more flexible subtyping
    - Using wildcards for more convenient bounds
    - Related digression: Java’s array subtyping
    - Java realities: type erasure
      - Unchecked casts
      - equals interactions
      - Creating generic arrays

Type erasure

All generic types become type Object once compiled
- Big reason: backward compatibility with ancient byte code
- So, at run-time, all generic instantiations have the same type

```java
List<String> lst1 = new ArrayList<String>();
List<Integer> lst2 = new ArrayList<Integer>();
lst1.getClass() == lst2.getClass() // true
```

Cannot use instanceof to discover a type parameter

```java
Collection<?> cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { // illegal
  …
}
```
Generics and casting

Casting to generic type results in an important warning

```java
List<?> lg = new ArrayList<String>();  // ok
List<String> ls = (List<String>) lg;   // warn
```

Compiler gives an unchecked warning, since this is something the runtime system will not check for you

Usually, if you think you need to do this, you’re wrong
- Most common real need is creating arrays with generic element types (discussed shortly), when doing things like implementing `ArrayList`.

Object can also be cast to any generic type

```java
public static <T> T badCast(T t, Object o) {
    return (T) o;   // unchecked warning
}
```

The bottom-line

- Java guarantees a `List<String>` variable always holds a (subtype of) the raw type `List`
- Java does not guarantee a `List<String>` variable always has only `String` elements at run-time
  - Will be true unless unchecked casts involving generics are used
  - Compiler inserts casts to/from `Object` for generics
    - If these casts fail due, hard-to-debug errors result: Often far from where conceptual mistake occurred
- So, two reasons not to ignore warnings:
  - You’re violating good style/design/subtyping/generics
  - You’re risking difficult debugging

Recall `equals`

```java
class Node {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node))  {
            return false;
        }
        Node n = (Node) obj;
        return this.data().equals(n.data());
    }
    ...
}
```

equals for a parameterized class

```java
class Node<E> {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<E>))  {
            return false;
        }
        Node<E> n = (Node<E>) obj;
        return this.data().equals(n.data());
    }
    ...
}
```

Equals for a parameterized class

```java
class Node<E> {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<?>) ) {  
            return false;
        }
        Node<E> n = (Node<E>) obj;
        return this.data().equals(n.data());
    }
    ...
}
```

Equals for a parameterized class

```java
class Node<E> {
    ...
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Node<?>))  {
            return false;
        }
        Node<?> n = (Node<?>) obj;
        return this.data().equals(n.data());
    }
    ...
}
```

Erasure: Type arguments do not exist at runtime

More erasure: At run time, do not know what `E` is and will not be checked, so don’t indicate otherwise

Works if the type of `obj` is `Node<Elephant>` or `Node<String>` or ...

Leave it to here to “do the right thing” if this and `n` differ on element type
Generics and arrays

```java
public class Foo<T> {
    private T aField;       // ok
    private T[] anArray;    // ok

    public Foo(T param) {
        aField = new T();    // compile-time error
        anArray = new T[10]; // compile-time error
    }
}
```

You cannot create objects or arrays of a parameterized type (Actual type info not available at runtime)

Necessary array cast

```java
public class Foo<T> {
    private T aField;
    private T[] anArray;

    @SuppressWarnings("unchecked")
    public Foo(T param) {
        aField = param;
        anArray = (T[])(new Object[10]);
    }
}
```

You can create variables of type T, accept them as parameters, return them, or create arrays by casting Object[]

- Casting to generic types is not type-safe, so it generates a warning
- Rare to need an array of a generic type (e.g., use `ArrayList`)

Generics clarify your code

```java
interface Map {
    Object put(Object key, Object value);
    ...
}
```

Some final thoughts...

```java
interface Map<Key,Value> {
    Value put(Key key, Value value);
    ...
}
```

- Generics usually clarify the implementation
- But sometimes ugly: wildcards, arrays, instantiation
- Generics always make the client code prettier and safer

Tips when writing a generic class

- Start by writing a concrete instantiation
  - Get it correct (testing, reasoning, etc.)
  - Consider writing a second concrete version
- Generalize it by adding type parameters
  - Think about which types are the same or different
  - The compiler will help you find errors
- As you gain experience, it will be easier to write generic code from the start