Generics

What if you could say that your code works with "some unspecified type," rather than a specific interface or class?

Generics implement the concept of *parameterized* types, which allow multiple types.

A class that holds a single object

```
class Automobile {}
public class Holder1 {
  private Automobile a;

  public Holder1(Automobile a) {
    this.a = a;
  }

  Automobile get() { return a; }
}
```

A class that holds an Object

```java
public class Holder2 {
  private Object a;
  public Holder2(Object a) { this.a = a; }

  public void set(Object a) { this.a = a; }
  public Object get() { return a; }

  public static void main(String[] args) {
    Holder2 h2 = new Holder2(new Automobile());
    Automobile a = (Automobile) h2.get();
    h2.set("Not an Automobile");
    String s = (String) h2.get();
    h2.set(1); // Autoboxes to Integer
    Integer x = (Integer) h2.get();
  }
}
```

Simple generic class

```
public class Holder3<T> {
  private T a;

  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }

  public static void main(String[] args) {
    Holder3<Automobile> h3 =
        new Holder3<Automobile>(new Automobile());
    Automobile a = h3.get(); // No cast needed
    // h3.set("Not an Automobile"); // Error
    // h3.set(1); // Error
  }
}
```

Generic Types and Methods

There can be:

- Generic classes
- Generic interfaces
- Generic methods
- Bounded generic types
- Generic wildcards

The core idea of Java generics: You tell it what type you want to use, and it takes care of the details.

Type erasure

Java generics are implemented using ***type erasure***. This means that any specific type information is erased when you compile your code.

How type erasure works?

```
public class Holder3<T> {
  private T a;

  public Holder3(T a) { this.a = a; }
  public void set(T a) { this.a = a; }
  public T get() { return a; }
}
…
Holder3<Long> h3 =
    new Holder3<Long>(1L);
Long n = h3.get();
```

$\Rightarrow$

```
public class Holder3 {
  private Object a;

  public Holder3(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }
}
…
Holder3 h3 = new Holder3(1L);
Long n = (Long) h3.get();
```

Compensating for erasure

```
public class Erased<T> {
  private final int SIZE = 100;

  public static void f(Object arg) {
    if(arg instanceof T) {} // Error
    T var = new T(); // Error
    T[] array = new T[SIZE]; // Error
    T[] array = (T[])new Object[SIZE]; // Unchecked
                           warning
  }
}
```

Generic class with two types

```
public class TwoTuple<A, B> {
  public final A first;
  public final B second;

  public TwoTuple(A a, B b) { first = a; second = b; }

  public String toString() {
    return "(" + first + ", " + second + ")";
  }
}
```

You cannot use primitives as type parameters!

```
TwoTuple<String, Integer> ttsi = new TwoTuple<String, Integer>("hi", 47);
```

but not

```
TwoTuple<double, int> ttdi = new TwoTuple<double, int>(47.0, 47);
```

Inheritance with generic classes

```
public class ThreeTuple<A, B, C> extends TwoTuple<A, B> {
  public final C third;

  public ThreeTuple(A a, B b, C c) {
    super(a, b);
    third = c;
  }

  public String toString() {
    return "(" + first + ", " + second + ", " +
        third +")";
  }
}
```

Generic interface

```
public interface Generator<T> { T next(); }

public class Fibonacci implements Generator<Integer> {
  private int count = 0;
  public Integer next() { return fib(count++); }

  ...

  public static void main(String[] args) {
    Fibonacci gen = new Fibonacci();
    for(int i = 0; i < 18; i++)
      System.out.print(gen.next() + " ");
  }
}
```

Generic Methods

```
public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f('c');
    gm.f(new String[] {“H”, “W”});
  }
}
```

The Syntax for Invoking a Generic Method

Generics have an optional syntax for specifying the type for a generic method. You can place the data type of the generic in angle brackets, < > , after the dot operator and before the method call.

```
    gm.<String>f(“a string object”);
    gm.<Integer>f(1);
    gm.<Double>f(1.0);
    gm.<Char>f('c');
    gm.<String[]>f(new String[] {“H”, “W”});
```

The syntax makes the code more readable and also gives you control over the generic type in situations where the type might not be obvious.

Leveraging type argument inference

```
public class Tuple {
  public static <A,B> TwoTuple<A,B> tuple(A a, B b) {
    return new TwoTuple<A,B>(a, b);
  }
  public static <A,B,C> ThreeTuple<A,B,C> tuple(A a,B b,C c) {
    return new ThreeTuple<A,B,C>(a, b, c);
  }
}
...
TwoTuple<String, Integer> ttsi = Tuple.tuple(“hi”, 47);
ThreeTuple<String,Integer,Double> ttsid=Tuple.tuple(“hi”,47,47.0);
```

Anonymous inner classes

```
public interface Generator<T> { T next(); }

class Customer {
  private Customer() {}

  public static Generator<Customer> generator() {
    return new Generator<Customer>() {
      public Customer next() { return new Customer(); }
    };
  }
}
...
Customer customer1 = Customer.generator().next();
Customer customer2 = Customer.generator().next();
```

Bounded Generic Types

Because erasure removes type information, the only methods you can call for an unbounded generic parameter are those available for Object. Bounds allow you to place constraints on the parameter types. Important effect is that you can call methods that are in your bound types.

```
interface HasColor {
    Color getColor();
}

class Colored<T extends HasColor> {
  T item;
  public Colored(T item) { this.item = item; }
  public T getItem() { return item; }

  // The bound allows you to call a method:
  public Color color() { return item.getColor(); }
}
```

Compound bounds

```
class Dimension { public int x, y, z; }

// Multiple bounds:
class ColoredDimension<T extends Dimension & HasColor> {
  T item;
  ColoredDimension(T item) { this.item = item; }
  T getItem() { return item; }
  Color color() { return item.getColor(); }
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}
```

Bounds and Inheritance

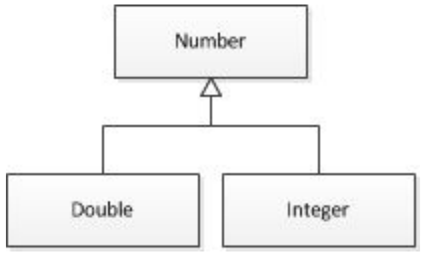
```
class HoldItem<T> {
  T item;
  HoldItem(T item) { this.item = item; }
  T getItem() { return item; }
}

class Colored2<T extends HasColor> extends HoldItem<T> {
  // some code here...
  Color color() { return item.getColor(); }
}

class ColoredDimension2<T extends Dimension & HasColor>
    extends Colored2<T> {
  // some code here...
  int getX() { return item.x; }
  int getY() { return item.y; }
  int getZ() { return item.z; }
}
```

Polymorphism and Generics

```
class Holder<T> {
  T item;
  void set(T item) { this.item = item; }
  T get() { return item; }
}
abstract class Parent { }
class Child extends Parent { }
class AnotherChild extends Parent { }
...
Holder<Child> h1 = new Holder<Child>(); // OK
Holder<Parent> h2 = new Holder<Parent>(); // OK
Holder<Parent> h3 = new Holder<Child>(); // Error

// because one could do this:
// h3.set(new AnotherChild());
```

Why Polymorphism doesn’t work

```
class Holder<T> {
  T[] items;
  int num = 0;
  void add(T item) { this.items[num++] = item;}
  T[] get() { return items; }
}
...
Holder<Integer> h = new Holder<Integer>();
h.add(1);
h.add(2);

Holder<Number> reg = h;
h.add(new Double(1.25));

Integer i3 = h.get()[2]
```



If polymorphism was allowed this would be legal

This is also legal since Double is a Number

h -> {Integer, Integer, Double}

Class case exception Double != Integer

So Double is a Number but Holder<Double> is not Holder<Number>

But you can do this:

This is how you can put different object types in parameterized Holder object:

```
Holder<Number> h = new Holder<Number>();
h.add(1);
h.add(2);
h.add(new Double(1.25));

Number i3 = h.get()[2]
```

Both Integer and Double are the Numbers

More Example

```
abstract class Animal { public abstract void check(); }
class Dog extends Animal {
  public void check() { S.o.p("Dog"); }
}
class Cat extends Animal {
  public void check() { S.o.p("Cat"); }
}
class AnimalDoctor {
  void checkAnimal(Holder<Animal> animal) {
    animal.get().check();
  }
  public static void main(String[] args) {
    Holder<Dog> dog = new Holder<Dog>();
    Holder<Cat> cat = new Holder<Cat>();

    AnimalDoctor doctor = new AnimalDoctor();
    doctor.checkAnimal(dog); // Error
    doctor.checkAnimal(cat); // Error
  }
}
```

Generic Wildcards (?)

The wildcard provides a polymorphic - like behavior for declaring generics.

- <?> , an unbounded wildcard
- <? extends *type*> , a wildcard with an upper bound
- <? super *type*> , a wildcard with a lower bound

Unbounded Wildcards

The *unbounded wildcard* represents any data type, similar to the < T > syntax.

```
public static void printList(List<?> list) {
  for(Object x : list) {
    System.out.println(x.toString());
  }
}
...
ArrayList<String> keywords = new ArrayList<String>();
kyewords.add("generic");
printList(keywords);
```

Data type is not required here

Use the ? in situations where you do not need a formal parameter type like < T >

Be careful

```
Holder<?> h = new Holder<String>();
h.add(new Object()); // compile time error
h.add(new String()); // compile time error

// one exception!
h.add(null); // null is member of every type
```

Working with unbounded wildcards we can only read data, not assign!

Upper - Bound Wildcards

Bounded wildcards put some restrictions on unknown type:

```
public static void printList(List<? extends Number> list){
  for(Number x : list) {
    System.out.println(x.doubleValue());
  }

  list.add(new Integer(3));  // compile error
}
```

Now we know that object is instance of Number

But we still don’t know exact type, so can’t modify list

More example

```
class AnimalDoctor {
  void checkAnimal(Holder<? extends Animal> animal) {
    animal.get().check(); // OK
    animal.set(new Cat()); // Error: we don't know exact
parameter type of animal
  }
  public static void main(String[] args) {
    Holder<Dog> dog = new Holder<Dog>();
    Holder<Cat> cat = new Holder<Cat>();

    AnimalDoctor doctor = new AnimalDoctor();
    doctor.checkAnimal(dog); // OK
    doctor.checkAnimal(cat); // OK
  }
}
```

Lower Bounded Wildcards

a *lower bounded* wildcard restricts the unknown type to be a specific type or a *super type* of that type

```
Holder<? super Integer> h = new Holder<Integer>();
h.add(new Integer(1));

Integer i1 = h.get();    // compilation error
            // get returns Object

Integer i2 = (Integer)h.get(); // OK
```

Lower bounded wildcards allow to modify but not read!!

Example

```
public static void addNumbers(List<? super Integer> list) {
  for (int i = 1; i <= 10; i++) {
    list.add(i);
  }
}

List<Integer> i = new ArrayList<Integer>();
List<Number> n = new ArrayList<Number>();
List<Object> o = new ArrayList<Object>();

addNumbers(i);
addNumbers(n);
addNumbers(o);
```

This works fine

More example

```
class AnimalDoctor {
  void checkAnimal(Holder<? super Dog> dog) {
    animal.get().check(); // Error!
                          // get() returns Object ref
    animal.set(new Dog()); // OK
  }
  public static void main(String[] args) {
    Holder<Dog> dog = new Holder<Dog>();
    Holder<Animal> animal = new Holder<Animal>();
    Holder<Cat> cat= new Holder<Cat>();

    AnimalDoctor doctor = new AnimalDoctor();
    doctor.checkAnimal(dog); // OK
    doctor.checkAnimal(animal); // OK
    doctor.checkAnimal(cat); //Error: Cat isn't super of
Dog
  }
}
```

What's the difference?

```
void check(Holder<?> holder) { }

void check(Holder<Object> holder) { }
```

There IS a huge difference!

```
class Main {
  void check(Holder<Object> obj) {
    obj.set(new Dog());
    obj.set(new Cat());
  }
  public static void main(String[] args) {
    Holder<Object> obj = new Holder<Object>();

    Main main = new Main();
    main.check(obj); // Only Holder<Object> goes here!
  }
}
```

There IS a huge difference!

```
class Main {
  void check(Holder<?> obj) {
    obj.set(new Dog()); // Error
            // Compiler isn't sure that it's really Dog Holder
  }
  public static void main(String[] args) {
    Holder<Dog> dog = new Holder<Dog>();
    Holder<Integer> integer = new Holder<Integer>();

    Main main = new Main();
    main.check(dog); // OK
    main.check(integer); // OK
  }
}
```

Which will compile?

```
1) List<?> list = new ArrayList<Dog>();
2) List<? extends Animal> aList = new ArrayList<Dog>();
3) List<?> foo = new ArrayList<? extends Animal>();
4) List<? extends Dog> cList = new ArrayList<Integer>();
5) List<? super Dog> bList = new ArrayList<Animal>();
6) List<? super Animal> dList = new ArrayList<Dog>();
```

Which will compile?

1) `List<?> list = new ArrayList<Dog>();`

2) `List<? extends Animal> aList = new ArrayList<Dog>();`

~~3)~~ `List<?> foo = new ArrayList<? extends Animal>();`

~~4)~~ `List<? extends Dog> cList = new ArrayList<Integer>();`

5) `List<? super Dog> bList = new ArrayList<Animal>();`

~~6)~~ `List<? super Animal> dList = new ArrayList<Dog>();`

Naming Conventions for Generics

- E for an element
- K for a map key
- V for a map value
- N for a number
- T for a generic data type

Use S , U , V , and so on for multiple types in the same class.

Summary

1) Generics let you enforce compile-time type safety.

```
Holder<String> h1 = new Holder<String>();
Holder h2 = new Holder();
String s = h1.get(); // no cast needed
String s = (String) h2.get(); // cast required
```

2) Generic type information does not exist at runtime — it is for compile-time safety only.

3) Polymorphic assignment don't apply to the generic type

```
Holder<Animal> h1 = new Holder<Dog>(); // Error
```

4) Wildcard syntax allows a generic method, accept subtypes (or supertypes) of the declared type of the method argument:

```
void foo(List<Dog> d) {} // can take only <Dog>
void foo(List<? extends Dog>) {} // take a <Dog> or <Collie>
void foo(List<? super Dog>) {} // take a <Dog> or <Animal>
```

5) When using a wildcard, List<? extends Dog>, the collection can be accessed but not modified.

Summary

6) The generics type identifier can be used in class, method, and variable declarations:

```
class Foo<t> { } // a class
T anInstance; // an instance variable
Foo(T aRef) {} // a constructor argument
void bar(T aRef) {} // a method argument
T baz() {} // a return type
```

7) You can use more than one parameterized type in a declaration:

```
public class UseTwo<T, X> { }
```

8) You can declare a generic method using a type not defined in the class:

```
public <T> T returnMe(T t) { return t; }
```

Reading in Russian

http://www.rsdn.ru/article/java/genericsinjava.xml